(5)

(19)日本国特許庁（ＪＰ） (12) 公開特許公報（A） (11)特許出願公開番号
特開2002－116916
（P2002－116916A）
(43)公開日 平成14年４月19日(2002.4.19)

(51)Int.Cl.'		識別記号	FI		テーマコード'(参考)
G06F	9/45		G06F 9/32	340B	5B013
	9/32	340	9/38	370X	5B033
	9/38	370	9/44	322F	5B081

審査請求 有 請求項の数15 OL （全 30 頁）

(21)出願番号 特願2000－304618(P2000－304618)

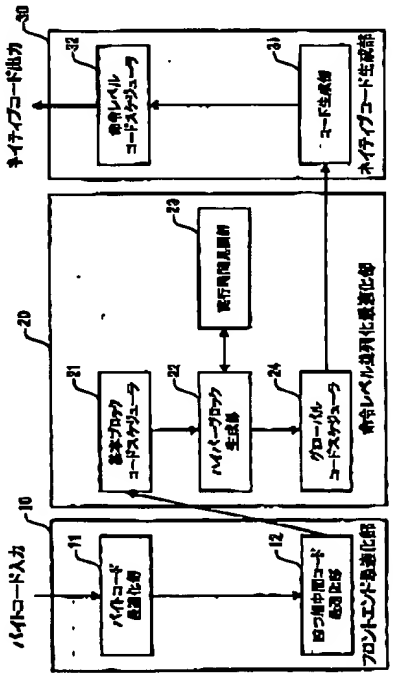
(22)出願日 平成12年10月４日(2000.10.4)

(71)出願人 390009531
インターナショナル・ビジネス・マシーンズ・コーポレーション
INTERNATIONAL BUSINESS MASCHINES CORPORATION
アメリカ合衆国10504、ニューヨーク州 アーモンク （番地なし）
(74)復代理人 100104880
弁理士 古部 次郎 （外４名）

最終頁に続く

(54)【発明の名称】 プログラムの最適化方法及びこれを用いたコンパイラ

(57)【要約】
【課題】 高速かつ適切に、プログラムをハイパーブロックに分割することにより、プログラムの実行効率を向上させることを可能とする。
【解決手段】 処理対象のプログラムに対し、基本ブロックを単位として実行時間を見積もる基本ブロック・コードスケジューラ２１と、基本ブロックを並列実行領域であるハイパーブロックにまとめるハイパーブロック生成部２２及び実行時間見積部２３とを備えたコンパイラであり、実行時間見積部２３は、基本ブロックの接続が条件分岐を伴う場合に、前記基本ブロック単位で見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、条件分岐部分を並列実行する場合の実行時間を見積もり、ハイパーブロック生成部２２は、並列実行する方が実行時間が短い場合、この部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短い場合、この部分をこの条件分岐にて接続された複数の並列実行領域に分割する。



【特許請求の範囲】

【請求項１】 プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行う最適化方法において、

処理対象である前記プログラムに対し、基本ブロックを単位として実行時間を見積もるステップと、

前記基本ブロックの接続関係を入れ子構造で表すネスト木を生成するステップと、

前記ネスト木のノードが条件分岐を伴う場合に、前記基本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、当該プログラムの条件分岐部分を並列実行する場合の、当該プログラムの当該ノード部分における実行時間を見積もるステップと、

前記見積もりにより、並列実行する方が実行時間が短い場合は、当該ノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短い場合は、当該ノードの複数の子ノードを複数の並列実行領域に分割するステップとを含むことを特徴とするプログラムの最適化方法。

【請求項２】 前記基本ブロックを単位として実行時間を見積もるステップは、

前記基本ブロックを単位として見積もられた実行時間に基づいて、さらに、前記基本ブロック内のプログラム部分におけるクリティカルパス長と、当該プログラム部分の平均並列度とを取得するステップを含む請求項１に記載のプログラムの最適化方法。

【請求項３】 前記ネスト木を生成するステップは、

前記基本ブロック間の依存関係を表す依存グラフを生成するステップと、

前記依存グラフから冗長な枝を取り除いた先行制約グラフを生成するステップと、

前記先行制約グラフのノードの接続関係を入れ子構造で表現することにより前記ネスト木を生成するステップとを含む請求項１に記載のプログラムの最適化方法。

【請求項４】 前記条件分岐部分の実行時間を判断するステップは、

前記子ノードにおいて実行可能な並列度ごとに、当該子ノードを並列実行した場合の実行時間の最大値を求めるステップと、

各並列度における前記実行時間の最大値のうち特定の値を前記条件分岐部分を並列実行する場合における実行時間と見積もるステップとを含む請求項１に記載のプログラムの最適化方法。

【請求項５】 前記条件分岐部分の実行時間を判断するステップは、

実行時間の判断に先立って、前記子ノードを構成する前記基本ブロックの命令レベルでの依存関係に基づいて、前記基本ブロックの実行時間に関する情報を修正するステップを含む請求項１に記載のプログラムの最適化方法。

【請求項６】 前記プログラムの並列実行領域を決定するステップは、

前記複数の子ノードを複数の並列実行領域に分割する場合に、ハードウェアが持つ並列度で各子ノードを並列実行した場合における当該各子ノードの実行時間を比較するステップと、

前記実行時間が最も短い子ノードを残して、他の子ノードを独立した並列実行領域とするステップとを含む請求項１に記載のプログラムの最適化方法。

【請求項７】 プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行う最適化方法において、

処理対象である前記プログラムの条件分岐部分を、プレディケート付き命令が実行可能でかつ命令レベルの並列実行が可能な計算機にて並列実行する場合の実行時間を見積もるステップと、

見積もられた前記実行時間が、前記条件分岐部分をそのまま実行した場合の実行時間よりも短い場合に、当該条件分岐部分を前記プレディケート付き命令による並列実行を行うように書き換えるステップとを含み、

前記並列実行時の実行時間を見積もるステップは、

前記プログラムに対し、基本ブロックを単位として実行時間を見積もり、当該実行時間に基づいて、各基本ブロック内のプログラム部分におけるクリティカルパス長と、当該プログラム部分の平均並列度とを求めるステップと、

前記条件分岐による分岐先である基本ブロックを、前記クリティカルパス長及び平均並列度の情報に基づいて、実行可能な並列度ごとに、当該基本ブロックを並列実行した場合の実行時間の最大値を求めるステップと、

各並列度における前記実行時間の最大値のうちの特定の値を前記条件分岐部分を並列実行する場合における実行時間と見積もるステップとを含むことを特徴とするプログラムの最適化方法。

【請求項８】 前記基本ブロックのクリティカルパス長及び平均並列度とを求めるステップは、前記基本ブロックを、隣り合う２辺の一方の値を前記クリティカルパス長とし、他方の値を前記平均並列度値とすると共に、クリティカルパス長に対応する辺が当該クリティカルパス長を下回らない範囲で変形可能とした矩形で表現するステップを含む請求項７に記載のプログラムの最適化方法。

【請求項９】 前記条件分岐部分の実行時間を見積もるステップは、実行時間の判断に先立って、前記基本ブロックの命令レベルでの依存関係に基づいて、前記基本ブロックの実行時間に関する情報を修正するステップを含む請求項７に記載のプログラムの最適化方法。

【請求項１０】 プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの

最適化を行うコンパイラにおいて、
処理対象である前記プログラムに対し、基本ブロックを単位として実行時間を見積もる第１のコードスケジューラと、
前記基本ブロックをまとめて並列実行領域であるハイパーブロックを生成するハイパーブロック生成部と、
前記プログラムにおける所定の領域を実行した場合の処理時間を見積もることにより、前記ハイパーブロック生成部による前記ハイパーブロックの生成を支援する実行時間見積部と、
生成された前記ハイパーブロックごとにコードスケジューリングを行う第２のコードスケジューラとを備え、
前記実行時間見積部は、
前記基本ブロックの接続関係を入れ子構造で表すネスト木の所定のノードが条件分岐を伴う場合に、前記基本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、当該プログラムの条件分岐部分を並列実行する場合の、当該プログラムの当該ノード部分における実行時間を見積もり、
前記ハイパーブロック生成部は、
前記実行時間見積部の見積もりにより、並列実行する方が実行時間が短いノードに関して、当該ノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短いノードに関して、当該ノードの複数の子ノードを複数の並列実行領域に分割することを特徴とするコンパイラ。
【請求項１１】　前記第１のコードスケジューラは、基本ブロックを単位として見積もられた実行時間に基づいて、さらに、前記基本ブロック内のプログラム部分におけるクリティカルパス長と、当該プログラム部分の平均並列度とを取得する、請求項１０に記載のコンパイラ。
【請求項１２】　前記実行時間見積部は、
前記子ノードにおいて実行可能な並列度ごとに、当該子ノードを並列実行した場合の実行時間の最大値を求め、各並列度における前記実行時間の最大値のうちの特定の値を前記条件分岐部分を並列実行する場合における実行時間と見積もる、請求項１０に記載のコンパイラ。
【請求項１３】　前記実行時間見積部は、実行時間の判断に先立って、前記子ノードを構成する前記基本ブロックの命令レベルでの依存関係に基づいて、前記基本ブロックの実行時間に関する情報を修正する、請求項１０に記載のコンパイラ。
【請求項１４】　コンピュータに実行させるプログラムを当該コンピュータの入力手段が読取可能に記憶した記憶媒体において、
処理対象であるプログラムに対し、基本ブロックを単位として実行時間を見積もる処理と、
前記基本ブロックの接続関係を入れ子構造で表すネスト木を生成する処理と、
前記ネスト木のノードが条件分岐を伴う場合に、前記基本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、当該プログラムの条件分岐部分を並列実行する場合のどちらが前記プログラムの当該ノード部分における実行時間が短くなるかを判断する処理と、
並列実行する方が実行時間が短いと判断した場合は、当該ノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短いと判断した場合は、当該ノードの複数の子ノードを複数の並列実行領域に分割する処理とを前記コンピュータに実行させることを特徴とする記憶媒体。
【請求項１５】　コンピュータに、
処理対象であるプログラムに対し、基本ブロックを単位として実行時間を見積もる処理と、前記基本ブロックの接続関係を入れ子構造で表すネスト木を生成する処理と、前記ネスト木のノードが条件分岐を伴う場合に、前記基本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、当該プログラムの条件分岐部分を並列実行する場合のどちらが前記プログラムの当該ノード部分における実行時間が短くなるかを判断する処理と、並列実行する方が実行時間が短いと判断した場合は、当該ノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短いと判断した場合は、当該ノードの複数の子ノードを複数の並列実行領域に分割する処理とを実行させるプログラムを記憶する記憶手段と、
前記記憶手段から前記プログラムを読み出して当該プログラムを送信する送信手段とを備えたことを特徴とするプログラム伝送装置。
【発明の詳細な説明】
【０００１】
【発明の属する技術分野】本発明は、コンピュータプログラムの最適化方法に関し、特にプログラムの条件分岐部分を並列実行に書き換える最適化を効果的に行うためにプログラムの最適化領域を編集する方法に関する。
【０００２】
【従来の技術】通常、プログラミング言語で記述されたプログラムのソースコードをコンパイルする際、コンピュータにおける実行速度の向上を図るために、当該プログラムの最適化を行っている。最適化の手法には種々の方法があるが、米国インテル社及び米国ヒューレット・パッカード社によるＩＡ－６４のアーキテクチャに対応したＣＰＵのように、ＶＬＩＷ（Very Long Instruction Word）による並列処理、すなわち、プレディケート付き命令が実行可能でかつ命令レベルの並列実行が可能なプロセッサ上では、分岐命令を削除して、分岐先命令群を相補的なプレディケートを付けて並列実行することができる（このように命令群を変換することを、以下では、ＩＦ変換と呼ぶ）。このＩＦ変換を行うことにより、命令数を減少したり、分岐予測の失敗を回避したり

することができるため、プログラムの実行効率を向上させることが可能である。

【０００３】しかし、ＩＦ変換は、この変換を行う命令の選び方によっては、かえって実行効率を低下させてしまう場合がある。その原因は、並列度のハードウェア限界の超過、レジスタープレッシャーの上昇、分岐先命令におけるクリティカルパス長のバランスの悪さ、実行可能性が低い命令の挿入などである。これら全ての原因を考慮に入れてプログラムの実行効率の高低を厳密に判断するには、各分岐命令に関して、ＩＦ変換する場合としない場合のそれぞれについてコードスケジュールを実行し、実際の命令サイクル数を見積もり、比較する必要がある。

【０００４】しかしながら、プログラム中の全ての分岐命令に対して、この命令サイクル数の見積もり及び比較を行うとすると、組み合わせの数が膨大となるため、現実的な計算時間で終了することができない。そのため、最適化を実行する領域（以下、この領域をハイパーブロックと称し、図示する場合には四辺形などを用いて示す）を適切に選択する必要がある。

【０００５】そこで従来は、ＩＦ変換を行うかどうかの判断を現実的な時間内で処理するため、（１）実行可能性が最も高いと予想される実行パス（以下、メイントレースと呼ぶ）中の分岐命令のみを必要に応じてＩＦ変換する方法や、（２）一旦、全ての分岐命令をＩＦ変換し最適化を施した後、リストスケジューリングの過程で必要に応じてＩＦ変換の逆方向の変換（以下、逆ＩＦ変換と呼ぶ）を実行して分岐命令を再生する方法を採っていた。

【０００６】これらの従来技術について、さらに説明する。

（１）メイントレース中の分岐命令のみを必要に応じてＩＦ変換する従来技術としては、文献１

S.A.Mahlke, R.E.Hank, R.A.Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock" in Proceedings of the 25th International Symposium on Microarchitecture, pp. 45-54, December 1992. に開示された技術がある。文献１に記載された従来技術は、どの領域を並列実行し、ＩＦ変換を実行することで性能向上が得られるかという課題について、発見的手法による一つの解を与えている。文献１によれば、まずメイントレースを特定し、このパスに対して無条件でＩＦ変換を実行する。次に、メイントレース以外のパス（以下、サブトレースと呼ぶ）それぞれについて、同じ並列実行領域に含めるかどうかを判断し、段階的にＩＦ変換を実行する領域を増大させてゆく。ある分岐命令に対してＩＦ変換を実行するかどうかは、以下の四つの要因を考慮して判断する。

１．サブトレースにパイプラインを乱す命令があるかどうか。

２．メイントレースに対するサブトレースの実行確率。

３．メイントレースに対するサブトレースの機械語命令数の比。

４．ハードウェアの並列実行能力の限界。

この方法によれば、メイントレースの分岐命令の数をｎとすると、ｎに比例する程度の計算量で、ＩＦ変換を行うかどうかの判断を終了することができる。

【０００７】（２）一旦、全ての分岐命令をＩＦ変換し最適化を施した後、リストスケジューリングの過程で必要に応じて逆ＩＦ変換を実行して分岐命令を再生する従来技術としては、文献２

D.I.August, W.W.Hwu, S.A.Mahlke, "A Framework for Balancing Control Flow and Predication" in Proceedings of the 30th International Symposium on Microarchitecture, December 1997. に開示された技術がある。文献２に記載された従来技術によれば、まず、プログラム全体を一つの並列実行領域とし、全ての分岐命令に対してＩＦ変換を行う。そして、ＩＦ変換後のプログラムに対して種々の最適化を施し、この後、選択的に逆ＩＦ変換を実行することによって、結果として選択的に分岐命令をＩＦ変換した状態を作る方法である。この方法は、コードスケジューラと協業して、各分岐命令に関して、逆ＩＦ変換した場合としない場合のそれぞれについて実行サイクル数を求める。そして、どちらの実行性能が高いかによって逆ＩＦ変換するかどうかを決定する。ただし、この方法を関数内の全ての分岐命令に対して適用すると、ＩＦ変換を行うかどうかを判断する場合と同様に、逆ＩＦ変換の対象となる命令群の組み合わせの数が膨大となる。そこで、リストスケジューラと協業し、クリティカルパスをスケジュールする場合にのみ逆ＩＦ変換を試みることによって、計算量を抑えている。文献２においては、分岐命令の数をｎとすると、２ｎ回程度スケジューリングを行う手法を提案している。なお、クリティカルパスとは、プログラム中の特定の範囲において、並列化できない一連の命令列のうちで最も長い命令列である。

【０００８】

【発明が解決しようとする課題】しかし、上述した従来技術において、（１）メイントレース中の分岐命令のみを必要に応じてＩＦ変換する技術は、メイントレースの実行効率を高めることはできるが、サブトレースの実行効率については特に考慮されていない。したがって、必ずしもプログラム全体の実行効率を向上させられるとは限らない。また、特に実行可能性の高いパスが無く、メイントレースを厳密に特定できない場合には、どのパスに対してＩＦ変換を行うかを決定することが困難である。そして、何らかの基準に基づいていずれかのパスに対してＩＦ変換を行ったとしても、他のパスの実行可能性も十分に高いため、プログラムの実行効率を十分に向上させることができない。

【0009】また、（2）一旦、全ての分岐命令をＩＦ変換し最適化を施した後、リストスケジューリングの過程で必要に応じて逆ＩＦ変換を実行して分岐命令を再生する技術は、リストスケジューリングの過程で逆ＩＦ変換の対象となるパスを選択しているが、依然として、所定のパスに対してＩＦ変換を行うかどうか（この場合は逆ＩＦ変換により分岐命令を再生するかどうか）を判断している。そして、この判断は、ＩＦ変換を行った場合と行わない場合とのそれぞれに対してコードスケジューリングを行って、比較するという枠組みであるため、計算量が大きい。すなわち、この従来技術においても、ＩＦ変換を行うかどうかを判断する場合と同様に、原理的に、ＩＦ変換を適用する領域の大きさと、コンパイルの速度との間にトレードオフが存在する。

【0010】以上のように、これらの従来技術は、現実的な汎用コンパイラ、特にＪａｖａにおけるJust In Time Compilerなどのようにコンパイル時間に制約がある言語処理系では、プログラム全体の十分な実行効率の向上を図ることができなかった。また、実際の汎用の言語処理系では、十分な実行時情報が得られないため、メイントレースのような最適化すべきパスを厳密に特定することは困難であり、この問題は一層深刻であった。

【0011】そこで、本発明は、限られた時間で、広い領域をハイパーブロックに分割することにより、ある程度実行可能性が高い多くのパスにおける実行効率を向上させることが可能な最適化方法を提供することを目的とする。

【0012】

【課題を解決するための手段】かかる目的のもと、本発明は、動的計画法に基づいた方法で、プログラムの所定の領域全体（命令列ａ）を並列実行した場合の最短処理時間の見積もりを、その領域の一部分（命令列ｂ、ｃなど）の最短処理時間の見積もりから再帰的に計算する。そして、命令列ａの実行効率と命令列ｂ、ｃを逐次に実行した場合の実行効率とを比較し、命令列ａの実行効率の方が悪くなる場合は、命令列ｂ及び命令列ｃを独立したハイパーブロックにする。これにより、プログラムにおける最適化したい領域全体を、複数のハイパーブロックに適切に分割する。

【0013】これを実現する本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行う最適化方法において、処理対象である前記プログラムに対し、基本ブロックを単位として実行時間を見積もるステップと、この基本ブロックの接続関係を入れ子構造で表すネスト木を生成するステップと、このネスト木のノードが条件分岐を伴う場合に、前記基本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、並列実行する場合の、このプログラムのこのノード部分における実行時間を見積もるステップと、この見積もりにより、並列実行する方が実行時間が短い場合は、このノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短い場合は、このノードの複数の子ノードを複数の並列実行領域に分割するステップとを含むことを特徴とする。

【0014】この最適化方法は、特にプレディケート付き命令が実行可能でかつ命令レベルの並列実行が可能な計算機にてプログラムを実行する場合に、条件分岐をそのまま実行するよりも命令レベルで並列実行する方が高速に処理できるならば、この条件分岐部分を並列実行するように書き換えることができる。

【0015】ここで、この基本ブロックを単位としてプログラムの実行時間を見積もるステップは、前記基本ブロックを単位として見積もられた実行時間に基づいて、さらに、この基本ブロック内のプログラム部分におけるクリティカルパス長と、このプログラム部分の平均並列度とを取得するステップを含む。また、この基本ブロックを、取得したクリティカルパス長及び平均並列度を辺とする矩形で表現し、さらに、この矩形を、クリティカルパス長に対応する辺がこのクリティカルパス長を下回らない範囲で変形可能とすることができる。基本ブロックについて取得されるクリティカルパス長は、この基本ブロック内部の命令列のうちで依存関係により逐次実行する必要のある最も長い部分の長さである。また、平均並列度は、基本ブロックに含まれる全ての命令の数（延べ実行時間に相当）をクリティカルパス長で割った値である。すなわち、基本ブロックの命令列を、クリティカルパス長を保持したまま並列に実行する場合に、どれだけの並列度が必要かを示す。このように平均並列度を定義した上で、複数の基本ブロックを並列に実行した場合に必要とされる並列度は、基本ブロックの平均並列度の線形和になると近似する。

【0016】また、このネスト木を生成するステップは、基本ブロック間の依存関係を表す依存グラフを生成するステップと、この依存グラフから冗長な枝を取り除いた先行制約グラフを生成するステップと、この先行制約グラフのノードの接続関係を入れ子構造で表現することによりこのネスト木を生成するステップとを含む。ネスト木のノードは、プログラムの基本ブロックと、シリーズスートと、パラレルスートとからなる。シリーズスートは基本ブロックまたはスートが依存関係を有して直列に接続されたものであり、パラレルスートは基本ブロックまたは他のスートが依存関係を持たずに並列に並んだものである。すなわち、基本ブロックまたはスートをシリーズスートまたはパラレルスートの形で括っていくことにより、このプログラムの入れ子構造を表現する。

【0017】また、ここで、条件分岐部分の実行時間を判断するステップは、この子ノードにおいて実行可能な並列度ごとに、この子ノードを並列実行した場合の実行時間の最大値を求めるステップと、各並列度における実

行時間の最大値のうち特定の値をこの条件分岐部分を並列実行する場合における実行時間と見積もるステップとを含む。

【００１８】この条件分岐部分の実行時間を判断するステップは、実行時間の判断に先立って、この子ノードを構成する基本ブロックの命令レベルでの依存関係に基づいて、この基本ブロックの実行時間に関する情報を修正するステップを含む。具体的には、シリーズスートを構成する基本ブロックにおいて、前に位置する基本ブロック内部における最後の命令以外の命令と、後ろに位置する基本ブロック内部における最初の命令とが依存関係を有する場合、このシリーズスートは基本ブロックを単に直列に連結するよりも短い長さ（このシリーズスートのクリティカルパス長）で実行できることとなる。

【００１９】さらにここで、このプログラムの並列実行領域を決定するステップは、この複数の子ノードを複数の並列実行領域に分割する場合に、ハードウェアが持つ並列度で各子ノードを並列実行した場合における各子ノードの実行時間を比較するステップと、この実行時間が最も短い子ノードを残して、他の子ノードを独立した並列実行領域とするステップとを含む。

【００２０】また、本発明は、これらの最適化方法を、コンピュータによるプログラムのコンパイルにおいて実行させるコンピュータプログラムとして作成し、このコンピュータプログラムを格納した記憶媒体や、このコンピュータプログラムを伝送する伝送装置として提供することができる。

【００２１】さらに、本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行うコンパイラにおいて、処理対象である前記プログラムに対し、基本ブロックを単位として実行時間を見積もる第１のコードスケジューラと、この基本ブロックをまとめて並列実行領域であるハイパーブロックを生成するハイパーブロック生成部と、このプログラムにおける所定の領域を実行した場合の処理時間を見積もることにより、このハイパーブロック生成部によるこのハイパーブロックの生成を支援する実行時間見積部と、生成されたハイパーブロックごとにコードスケジューリングを行う第２のコードスケジューラとを備え、この実行時間見積部は、この基本ブロックの接続関係を入れ子構造で表すネスト木の所定のノードが条件分岐を伴う場合に、前記基本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、このプログラムの条件分岐部分を並列実行する場合の、当該プログラムの当該ノード部分における実行時間を見積もり、このハイパーブロック生成部は、実行時間見積部の見積もりにより、並列実行する方が実行時間が短いノードに関して、当該ノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短いノードに関して、当該ノードの複数の子ノードを複数の並列実行領域に分割することを特徴とする。

【００２２】ここで、この第１のコードスケジューラは、基本ブロックを単位として見積もられた実行時間に基づいて、さらに、この基本ブロック内のプログラム部分におけるクリティカルパス長と、このプログラム部分の平均並列度とを取得する。また、この実行時間見積部は、実行時間の判断に先立って、この子ノードを構成する基本ブロックの命令レベルでの依存関係に基づいて、この基本ブロックの実行時間に関する情報を修正する。

【００２３】

【発明の実施の形態】以下、添付図面に示す実施の形態に基づいて、この発明を詳細に説明する。まず、本発明の概要を説明する。上述したように、本発明は、プログラムのコンパイル時における最適化において、プログラムの所定の領域を適切なハイパーブロックに分割する。

【００２４】具体的には、まず、処理対象であるプログラムを変形可能に定義された基本ブロックに分割し、基本ブロック間の制御依存関係及びデータ依存関係に基づいて、ＰＤＧ（Program Dependence Graph：依存グラフ）を作る。ここで、基本ブロックとは、ストレートコード、すなわちコントロールフローが途中に入ることもなく、途中から出ることもないようなコード列の範囲をブロックで示したものである。本発明では、後述のように基本ブロックを構成する命令列のクリティカルパス長を下回らない範囲で、変形可能に基本ブロックを定義する。次に、作成されたＰＤＧをシリーズパラレルグラフに変換し、このシリーズパラレルグラフを基本ブロック間の先行制約グラフとして用いる。そして、このシリーズパラレルグラフにおける各基本ブロック内に対してコードスケジューリングを実行し、メモリ依存を超える投機的命令移動などの最適化を実行する。なお、シリーズパラレルグラフの定義については後述する。

【００２５】次に、シリーズパラレルグラフにおける各基本ブロックに関して、クリティカルパス長、平均の並列度、消費するハードウェアリソースなどの情報を求めておき、これらの値を基に、基本ブロックの領域全体の最短処理時間を再帰的に見積もる。この過程で、所定の領域について、並列実行より逐次実行の方が処理速度が速いと判断された場合は、当該領域に含まれる部分領域それぞれを独立したハイパーブロックとし、当該ハイパーブロックの中の全ての条件分岐命令をＩＦ変換する。これらの仕組みは、動的計画法を用いることにより、基本ブロックの数に比例する計算時間で終了することができる。

【００２６】図１は、本発明の実施の形態におけるコンパイラの構成を説明する図である。図１に示すコンパイラは、ＪａｖａのJust In Time Compilerである。本実施の形態では、本発明をJust In Time Compilerに適用する場合を例として説明するが、他の種々のプログラム

言語で記述されたプログラムに対するコンパイラに適用できることは言うまでもない。

【００２７】図１を参照すると、本実施の形態におけるコンパイラは、フロントエンド最適化部１０と、命令レベル並列化最適化部２０と、ネイティブコード生成部３０とを備える。フロントエンド最適化部１０は、Ｊａｖａにおける処理対象のプログラムのバイトコードを入力し、バイトコードレベルでの最適化を行うバイトコード最適化部１１と、バイトコード最適化部１１により最適化を施されたバイトコード（四つ組中間コード）に対して四つ組中間コードレベルでの最適化を行う四つ組中間コード最適化部１２とを備える。また、四つ組中間コード最適化部１２では、処理対象であるプログラムを基本ブロックに分割する。

【００２８】命令レベル並列化最適化部２０は、フロントエンド最適化部１０により四つ組中間コードレベルまでの最適化が行われたプログラムに対して、基本ブロックレベルでのコードスケジューリングを行う基本ブロック・コードスケジューラ２１と、当該基本ブロック間の依存関係に基づいてＰＤＧを作成し、このＰＤＧをハイパーブロックに適切に分割するためのハイパーブロック生成部２２及び実行時間見積部２３と、ハイパーブロックに分割されたプログラム全体に対するコードスケジューリングを行うグローバル・コードスケジューラ２４とを備える。

【００２９】ネイティブコード生成部３０は、命令レベル並列化最適化部２０によりコードスケジューリングが行われたプログラムをネイティブコードに変換するコード生成部３１と、コード生成部３１により生成されたネイティブコードに対して命令レベルでのコードスケジューリングを行う命令レベル・コードスケジューラ３２とを備える。

【００３０】以上の構成のうち、フロントエンド最適化部１０とネイティブコード生成部３０とは、従来のJust In Time Compilerにおけるフロントエンド最適化部１０及びネイティブコード生成部３０と同様である。したがって、本実施の形態は、命令レベル並列化最適化部２０において、基本ブロックレベルでのコードスケジューリングを行い、ハイパーブロック生成部２２及び実行時間見積部２３において、命令列の実行時間を見積もりながらハイパーブロックを生成した上で、全体的なコードスケジューリングを行う点に特徴がある。また、図１に示すJust In Time Compiler以外の、他のプログラミング言語におけるコンパイラにおいても、ソースプログラムを入力して構文解析を行い中間コードを生成した後に、図１の命令レベル並列化最適化部２０に相当する機能ブロックにより、同様の命令レベルでの並列化による最適化処理を行うことができる。

【００３１】なお、図１に示す各構成要素は、コンピュータプログラムにより制御されたＣＰＵにて実現される仮想的なソフトウェアブロックである。ＣＰＵを制御する当該コンピュータプログラムはＣＤ－ＲＯＭやフロッピー（登録商標）ディスクなどの記憶媒体に格納したり、ネットワークを介して伝送したりすることにより提供される。

【００３２】次に、命令レベル並列化最適化部２０の動作について説明する。本実施の形態は、処理対象であるプログラム中の命令間の制御依存関係及びデータ依存関係を示すＰＤＧ（Program Dependence Graph）をシリーズパラレルグラフに変換し、命令列の実行時間の見積もりを再帰的に行うことにより、適切なハイパーブロックを生成する。以下では、命令レベル並列化最適化部２０の動作について、まず概略的に説明し、次に具体的な手法について説明する。

【００３３】まず、本実施の形態で用いる基本ブロックについて詳細に説明する。図３９は、処理対象のプログラムの構成を基本ブロックで表現した例を示す図である。図３９に示した基本ブロックＤ、Ｅの組のように互いに依存がない基本ブロックの集合（パラレルスートと呼ぶ、定義は後述）の場合、並列に実行しても、逐次に実行しても、プログラムの正しさは保証されているので、どちらで実行されると領域全体の結果が良いかどうかを決定する必要がある。この決定は、単純には、領域中の全てのパラレルスートを、逐次、並列のどちらかで実行したときの全ての組み合わせに対して、全体の処理時間と必要な並列度を求めて比較する方法で実現できる。その際、全体の処理時間と必要な並列度を求めるには、単純には、パラレルスートの実行時間を構成要素の実行時間の合計ｄと論理的な並列度の最大値ｗとで表すことができる。図４０は、パラレルスートの実行時間を構成要素のパラメータ（ｄ、ｗ）で表現できることを説明する図である。

【００３４】しかし、この方法には、次の三つの問題点がある。１．基本ブロックは、上述したように、ストレートコード、すなわちコントロールフローが途中に入ることもなく、途中から出ることもないようなコード列の範囲をブロックで示したものである。したがって、本来的には変形不能の矩形領域ではなく、実際には（ある制約の下で）柔軟に移動可能な命令の集合であるが、上の基本ブロックの定義ではこの点が考慮されていない。２．スートを構成する際に図４０に示した「すきま」ができるため、処理時間と並列度の見積もり精度が悪い。３．領域中のパラレルスートを並列処理する場合と逐次処理する場合の全ての組み合わせを尽くす方法で実行時間を見積もるため、計算時間が基本ブロックの数に応じて指数関数的に増加する。

【００３５】そこで、本実施の形態では、各基本ブロックを、クリティカルパス長を下回らないという前提の下で、縦方向（処理時間が伸びる方向）に伸ばせるようにモデル化する。このようにモデル化された基本ブロック

によるパラレルスートの例を図４１に示す。図４１に示すように、パラレルスートは、基本ブロックＤと基本ブロックＥとを変形させながら処理時間と平均並列度を見積もることができる。これにより、より実際のプログラムの構成に即した見積もりを行うことが可能となる。また、基本ブロックを変形可能としたことにより、上述したスートにおける「すきま」を解消することもできる。さらに、基本ブロックの横幅である平均並列度は、基本ブロックにおける命令列の延べ実行時間とクリティカルパス長に基づいて単純なモデル化により算出された値であるため、図４２に示すような基本ブロック内部における隙間も、複数の基本ブロックを扱うことによってある程度相殺されることが期待される。

【００３６】次に、このモデルに従った最短処理時間の見積もりを、基本ブロックの数に応じて計算時間が指数関数的に増加することを回避して高速に実現するために、動的計画法を使った実現方法を説明する。本実施の形態では、所定の領域の最短処理時間の見積もりを、当該領域における部分領域の最短処理時間の見積もりから再帰的に求めていく。例えば、図３９の基本ブロックＤ、Ｅの処理時間の見積もりから、それらで構成されるパラレルスートの実行時間を見積もる。この再帰処理の１ステップを高速に実現することで、全体として高速な実行を可能とする。この処理を実現するには、基本ブロックだけでなく、パラレルスート、実行依存がある基本ブロックの集合（シリーズスートと呼ぶ、定義は後述）を、変形可能な矩形領域とみなせなければならない。これを満たすため、基本ブロック、パラレルスート、シリーズスートに関して、ハードウェアの並列度Ｗに対する、それぞれの可能な並列度ｗ（１≦ｗ≦Ｗ）に対する最短処理時間の見積データを再帰的に計算していく。

【００３７】基本ブロック、パラレルスート、シリーズスートの各場合について、上記のデータを見積もる方法を簡単に述べる。基本ブロックに対しては、基本ブロック・コードスケジューラ２１によるコードスケジューリングを実行し、延べ実行時間を求める。そして、この延べ実行時間を並列度ｗで割った値を、当該基本ブロックの最短処理時間の見積もりとする。ただし、本実施の形態における基本ブロックオン定義より、クリティカルパス長は下回らないものとする。シリーズスートに対しては、構成要素の並列度ｗにおける処理時間を、単純に加算する。パラレルスートに対しては、図４０のイメージどおり、１≦ｗ≦Ｗにおける各並列度ｗについて、ｗ１＋ｗ２＝ｗの条件を満たしながらｗ１を変化させ、各構成要素を並列実行した場合の実行時間の最大値を求め、得られた最大値の最小値を当該スートの最短処理時間の見積もりとする。これらの手続き、すなわち再帰の１ステップは、最長でも$W^2/2$に比例する計算時間で終了する。結局、基本ブロックの数をｎとすると、$n \times W^2$に比例した計算時間で全体の処理が終了する。

【００３８】次に、命令レベル並列化最適化部２０による動作の具体的な手法について説明する。本実施の形態では、上記の動的計画法を使ったハイパーブロックの生成手法として、基本ブロックの構成のみを用いた第１の手法と、基本ブロック内の命令レベルの依存関係をも考慮した第２の手法とを提案する。

【００３９】第１の手法は、基本ブロックを、論理的な並列度の平均とクリティカルパス長で表される矩形領域にモデル化する。複数の基本ブロックにおける逐次実行時間の見積もりは、クリティカルパスの処理時間の和を下回らないとする。また、複数の基本ブロックを並列に実行した場合に必要とされる並列度は、基本ブロックの平均並列度の線形和になると近似する。第２の手法は、基本ブロックを、論理的な並列度の平均とクリティカルパス長で表される矩形領域にモデル化すると共に、当該基本ブロック内の命令間における依存関係（Dependence Path）の情報を持たせる。複数の基本ブロックの逐次実行時間の見積もりは、Dependence Pathを再構成してクリティカルパス長を再計算し、その値を下回らないとする。複数の基本ブロックを並列に実行した時必要とされる並列度は、第１の手法と同様に扱う。基本ブロックを、平均並列度とクリティカルパス長で表される矩形領域に近似することにより、複数の基本ブロックを並列に実行した場合に必要とされる並列度は、基本ブロックの平均並列度の線形和になると近似することができ、これにより、高速動作を実現することができる。以下、手法ごとに詳細に説明する。なお、以下の説明において用いられる記号を図７において定義する。図７の各定義において、nodeはノード一般を示す。したがって、以下の説明において、特に所定のノードにおける子ノードを指す場合には、nodeの代わりにchild1などと記述する場合もある。

【００４０】〔第１の手法〕命令レベル並列化最適化部２０は、初期的に、フロントエンド最適化部１０により前処理としての最適化を施されたプログラム（中間コード）を入力し、実行頻度情報やプログラムの構造に基づいて並列実行する領域を決定する。例えば、ある程度以上の実行頻度を持ち、ループを含まない領域を並列実行する領域とすることができる。ここでは、図２に示したプログラム領域を処理対象とする。図２は、処理対象であるプログラムの最適化処理を行う領域の制御フローグラフと当該部分の命令列のリストである。

【００４１】基本ブロック・コードスケジューラ２１は、当該領域内の基本ブロック毎に、あいまいなメモリ依存の解消を伴うコードスケジューリングを行う。これにより、各基本ブロック（図７におけるnodeとする）について、node.clと、node.allとを見積もることができる。

【００４２】図３は、図２に示したプログラム領域において、node.clを縦の長さとし、node.allを面積とする

矩形領域で基本ブロックを表した図である。また、基本ブロックの横の長さ、すなわち面積（node.all）を縦の長さ（node.cl）で割った値を平均並列度と呼ぶ。例えば、図２によると、ノードＡには、「imove a, 1」「imove b, 1」「imove c, 3」「iadd a, 1」「isub b, 1」「cmpjmp-eq a, 1, C」という６個の命令が含まれる。このうち、「imove a, 1」「iadd a, 1」「cmpjmp-eq a, 1, C」は依存関係があるので、並列に実行できない。全部で６個の命令のうち、３個が逐次実行されるので、これがクリティカルパス長となる。したがって、図３に示すようにノードＡは、縦の長さが３、横の長さが２（＝６／３）の矩形として表現される。

【００４３】次に、ハイパーブロック生成部２２は、図３に示す基本ブロック間の制御依存関係及びデータ依存関係を表すＰＤＧを作成する。そして、作成されたＰＤＧから先行制約グラフとして冗長な枝を取り除き、シリーズパラレルグラフに変換する。ここで、シリーズパラレルグラフとは、次の三つの条件を満たすグラフである。

（１）単独のノードはシリーズパラレルグラフである。

（２）シリーズパラレルグラフを二つ直列に接続したものはシリーズパラレルグラフである。

（３）シリーズパラレルグラフを二つ並列に接続したものはシリーズパラレルグラフである。

また、シリーズパラレルグラフの構成は、シリーズスートとパラレルスートの２種類の部分に分けることができる。シリーズスートとは、直列に連結（この連結をシリーズコネクションと呼ぶ）された、依存関係のある一連のシリーズスート、パラレルスート及びノードの集合である。また、パラレルスートとは、並列に連結（この連結をパラレルコネクションと呼ぶ）された、互いに依存関係のないシリーズスート、パラレルスート及びノードの集合である。図８は、ＰＤＧをシリーズパラレルグラフに変換するアルゴリズムを示す擬似プログラムである。図示のアルゴリズムによる手続きは、現実的にはＰＤＧのノードの数に比例した計算時間で終了する。

【００４４】図４は、図３から作成されたＰＤＧを示す図、図５は、図４のＰＤＧから変換されたシリーズパラレルグラフを示す図である。図４、５を参照すると、例えば、図４におけるノードＡからノードＤへのデータ依存は、ノードＡからノードＣへの依存とノードＣからノードＤへの依存が存在するため、冗長とみなされ、図５に示すように取り除かれている。

【００４５】次に、ハイパーブロック生成部２２は、上述のシリーズパラレルグラフから、シリーズスートとパラレルスートの入れ子関係を表すシリーズパラレルネスト木を生成する。シリーズパラレルネスト木とは、次のように定義されるノードとエッジとを持つ木構造である。

ノード：シリーズパラレルグラフ中の全てのシリーズスートまたは全てのパラレルスートまたは全てのノードの集合。

エッジ：所定のシリーズスートに対し、シリーズコネクションのみで連結された一連のシリーズスート、パラレルスートまたはノードがある場合、この所定のシリーズスートから当該一連のスートまたはノードに対して張られるエッジ。または、所定のパラレルスートに対し、パラレルコネクションのみで連結された一連のシリーズスート、パラレルスートまたはノードがある場合、この所定のパラレルスートからその一連のスートまたはノードに対して張られるエッジ。図９は、シリーズパラレルグラフからシリーズパラレルネスト木を生成するアルゴリズムを示す疑似プログラムである。図示のアルゴリズムによる手続きは、実践的にはシリーズパラレルグラフのノード数に比例した時間で終了する。ただし、図示のアルゴリズムは、単純のため、パラレルスートが３以上の葉ノードを持つことを許しているが、葉ノードを２つだけ持つように変更することも容易に可能である。図６は、図５のシリーズパラレルグラフから生成されたシリーズパラレルネスト木を示す図である。なお、図５、６において、実線で示したノードはシリーズスートを示し、破線で示したノードはパラレルスートを示す。

【００４６】次に、ハイパーブロック生成部２２は、実行時間見積部２３を用いて、シリーズパラレルネスト木における各ノードの実行時間を再帰的に見積もり、その結果に基づいてハイパーブロック選択処理を実行する（図１０参照）。このハイパーブロック選択処理により、シリーズパラレルネスト木の各ノードに、独立したハイパーブロックとして扱うか否かを示す情報が付される。以下、ハイパーブロック選択処理について、詳細に説明する。

【００４７】図１１は、ハイパーブロック選択処理の全体的な動作の流れを示すフローチャートである。また、図１２は、図１１に対応する動作のアルゴリズムを示す疑似プログラムである。図１１を参照すると、まず、シリーズパラレルネスト木のノードの一つを処理対象とし（ステップ１１０１）、当該ノードの属性を調べ、パラレルスートか、シリーズスートか、基本ブロック（単一ノード）かを判断する（ステップ１１０２）。そして、当該ノードがパラレルスートであれば、実行時間見積部２３に処理を渡して当該パラレルスートの実行時間見積処理を行う（ステップ１１０３）。また、当該ノードがシリーズスートであれば、当該ハイパーブロック選択処理を当該ノードの子ノード（ここでは子ノード１、子ノード２の二つ）に再帰的に適用した後（ステップ１１０４）、当該シリーズスートの実行時間評価処理を行う（ステップ１１０５）。さらに、当該ノードが基本ブロック（単一ノード）であれば、単一ノードの実行時間評価処理を行う（ステップ１１０６）。

【００４８】ステップ１１０３によるパラレルスートの

実行時間見積処理では、並列度Wのときの所定のパラレルスート（ｘ）の処理時間を求めたい場合、パラレルスート（ｘ）を構成するスート（ｙ，ｚ）について、W＝ｗ１＋ｗ２の関係でｗ１を変化させながら、並列度ｗ１のときのスート（ｙ）の最短処理時間及び並列度ｗ２のときのスート（ｚ）の最短処理時間の最大値を求める。そして、ｗ１の値ごとに得られた最大値のうちの最も小さい値を、並列度Wのときのパラレルスート（ｘ）の最短処理時間として見積もる。ここで、パラレルスートは、各ノードをまとめて一つのハイパーブロックを構成することによって、処理時間を長くしてしまうことがあり得る。そこで、パラレルスート中のノードをまとめてハイパーブロックを構成するか否かの判断を行う。具体的には、分岐命令削減による利得を並列実行のオーバーヘッドが下回っているかどうかを判断する。そして、構成しないと判断された場合は、各ノードを分けて別のハイパーブロックを構成するという意味の印を付ける（node.hyper_block = TRUE（図７））。この際、どちらのスートを独立したハイパーブロックにするかの判断は、状況に応じて種々の手法で行うことができる。例えば、単純にクリティカルパスが長い方を独立したハイパーブロックとしても良いし、実行確率がわかっている場合は、実行確率の低い方を独立したハイパーブロックとすることもできる。

【００４９】図１３は、図１１におけるステップ１１０３のパラレルスートに対する実行時間見積処理の内容を説明するフローチャートである。図１３を参照すると、まず、処理対象であるノードの子ノード（ここでは子ノード１、子ノード２の二つ）に当該ハイパーブロック選択処理を再帰的に適用する（ステップ１３０１）。そして、パラレルスートの実行時間評価処理（ステップ１３０２）及び利得見積処理（ステップ１３０３）を行った後、得られた利得の値が０よりも大きいか否かを判断する（ステップ１３０４）。利得が０以下である場合は、ノード選択処理を行う（ステップ１３０５）。利得が０よりも大きい場合は、子ノード１と子ノード２の面積（すなわち、node.all：当該ノードの延べ実行時間）の和を親ノード（すなわち、初めに処理対象としたノード）の面積とする（ステップ１３０６）。

【００５０】図１４は、図１３におけるステップ１３０２の実行時間評価処理の内容を説明するフローチャートである。また、図１５は、図１４の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図１４を参照すると、まず、この処理の中でのみ有効な自由変数ｗ１の値を１に初期化する（ステップ１４０１）。また、この処理の中でのみ有効な自由変数ｍｉｎの値を無限大に初期化する（ステップ１４０２）。そして、変数ｗ１とハードウェアの並列度Wとを比較し、同じかどうか調べる（ステップ１４０３）。変数ｗ１が並列度Wと同じでなければ、次に、この処理の中でのみ有効な自由変数ｗ２を０に初期化し（ステップ１４０４）、この変数ｗ２と変数ｗ１とを比較して同じかどうか調べる（ステップ１４０５）。変数ｗ１と変数ｗ２とが同じでない場合、次に、子ノード１が並列度ｗ２のときの最短処理時間と、子ノード２が並列度（ｗ１－ｗ２）のときの最短処理時間とを比較し、大きい方の値を求める（ステップ１４０６）。そして、得られた値と、変数ｍｉｎの値とを比較し、小さい方の値を変数ｍｉｎの新たな値とする（ステップ１４０７）。この後、変数ｗ２の値を１増加し（ステップ１４０８）、ステップ１４０５の比較に戻る。一方、変数ｗ１と変数ｗ２とが同じである場合は、現時点での変数ｍｉｎの値を親ノード（すなわち、初めに処理対象としたノード）が並列度ｗ１のときの最短処理時間とする（ステップ１４０９）。そして、変数ｗ１の値を１増加し（ステップ１４１０）、ステップ１４０２に戻る。そして、ステップ１４０３の比較において、変数ｗ１と並列度Wとが同じであれば、処理を終了する（ステップ１４０３）。

【００５１】以上の動作により、並列度Wのときのパラレルスートの最短処理時間の見積もりが得られる。実際のＣＰＵ（ハードウェア）における並列度Wは一桁程度の小さな数値なので、この実行時間評価処理はノード数に比例する時間で終了することができる。

【００５２】図１６は、図１３におけるステップ１３０３の利得見積処理の内容を説明するフローチャートである。また、図１７は、図１６の利得見積処理の動作アルゴリズムを示す疑似プログラムである。図１６を参照すると、図１４、１５に示した実行時間評価処理により得られたパラレルスートの最短処理時間の見積もりに基づいて、子ノード１と子ノード２とを条件分岐としてそのまま実行した場合の処理時間と、この条件分岐をプレディケート付き命令で並列実行した場合の処理時間との差を利得として求める（ステップ１６０１）。

【００５３】図１８は、図１３におけるステップ１３０５のノード選択処理の内容を説明するフローチャートである。また、図１９は、図１８のノード選択処理の動作アルゴリズムを示す疑似プログラムである。図１８を参照すると、まず、ハードウェアの並列度Wのときの子ノード１及び子ノード２の最短処理時間を比較する（ステップ１８０１）。子ノード１の最短処理時間が子ノード２の最短処理時間以下である場合、子ノード２に対して当該子ノード２を独立したハイパーブロックとして実行するという意味の属性を付する（ステップ１８０２）。そして、子ノード１の最短処理時間の情報、依存関係を示す情報、及び延べ実行時間の情報を親ノード（すなわち、最初に処理対象としたノード）に複写する（ステップ１８０３）。これにより、当該親ノードは、子ノード１のみから構成されることとなる。一方、ステップ１８０１の判断において、子ノード１の最短処理時間の方が子ノード２の最短処理時間よりも大きい場合、子ノード

１に対して当該子ノード１を独立したハイパーブロックとして実行するという意味の属性を付する（ステップ１８０４）。そして、子ノード２の最短処理時間の情報、依存関係を示す情報、及び延べ実行時間の情報を親ノード（すなわち、最初に処理対象としたノード）に複写する（ステップ１８０５）。これにより、当該親ノードは、子ノード２のみから構成されることとなる。

【００５４】以上のようにして、シリーズパラレルネスト木のパラレルスートに対する実行時間見積処理を終了する。これにより、シリーズパラレルネスト木中のパラレルスートに関して、当該パラレルスートを構成するノードを一つのハイパーブロックにまとめるか、または別のハイパーブロックとして分岐させるかを示す情報と、当該一つまたは二つのハイパーブロックにおける実行時間の見積もりが得られ、当該ハイパーブロックに付されることとなる。

【００５５】図１１のステップ１１０５によるシリーズスートの実行時間評価処理では、当該シリーズスートを構成するノードの最短処理時間を単純に加えることにより、当該シリーズスートにおける実行時間を粗く見積もる。そしてさらに、平均並列度ｗに対して当該実行時間が最短（最短処理時間）となるように補正する。

【００５６】図２０は、図１１におけるステップ１１０５のシリーズスートに対する実行時間評価処理の内容を説明するフローチャートである。また、図２１は、図２０の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図２０を参照すると、まず、処理対象であるノードの延べ実行時間を子ノード（ここでは子ノード１、子ノード２の二つ）の延べ実行時間の和とする（ステップ２００１）。次に、この処理の中でのみ有効な自由変数ｗの値を１に初期化する（ステップ２００２）。そして、変数ｗとハードウェアの並列度Ｗとを比較し、同じかどうか調べる（ステップ２００３）。変数ｗが並列度Ｗと同じでなければ、次に、処理対象であるノードの並列度ｗのときの最短処理時間を、子ノード１及び子ノード２の並列度ｗのときのそれぞれの最短処理時間の和とする（ステップ２００４）。そして、処理対象であるノードの最短処理時間に変数ｗを乗じた値が、子ノード１及び子ノード２の延べ実行時間とを比較し、同じかどうか調べる（ステップ２００５）。すなわち、変数ｗが処理対象であるノードの平均並列度となっているかどうかを確認する。処理対象であるノードの最短処理時間に変数ｗを乗じた値が、子ノード１及び子ノード２の延べ実行時間とは異なっている場合、当該処理対象であるノードの平均並列度に対する最短処理時間をステップ２００４で求めた値とする（ステップ２００６）。処理対象であるノードの最短処理時間に変数ｗを乗じた値が、子ノード１及び子ノード２の延べ実行時間と同じである場合、またはステップ２００６の終了後、変数ｗの値を１増加し（ステップ２００７）、ステップ２００３に戻る。そして、ステップ２００３の比較において、変数ｗが並列度Ｗと同じであれば、処理を終了する（ステップ２００３）。

【００５７】図１１のステップ１１０６による単一ノードの実行時間評価処理では、クリティカルパス長を下回らない限り、計算総量を並列度で割った値で実行できると見積もる。図２２は、図１１におけるステップ１１０６の単一ノードに対する実行時間評価処理の内容を説明するフローチャートである。また、図２３は、図２２の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図２２を参照すると、まず、この処理の中でのみ有効な自由変数ｗを１に初期化する（ステップ２２０１）。そして、変数ｗとハードウェアの並列度Ｗとを比較し、同じかどうか調べる（ステップ２２０２）。変数ｗが並列度Ｗと同じでなければ、次に、処理対象であるノードの最短処理時間を、基本ブロック中の延べ実行時間を変数ｗで割った値とする（ステップ２２０３）。そして、得られた値が基本ブロックのクリティカルパス長を下回ったかどうかを判断する（ステップ２２０４）。ステップ２２０３で算出された処理対象であるノードの最短処理時間が、基本ブロックのクリティカルパス長を下回っている場合、当該ノードの最短処理時間を基本ブロックのクリティカルパス長とする（ステップ２２０５）。ステップ２２０３で算出された処理対象であるノードの最短処理時間が、基本ブロックのクリティカルパス長を下回っていない場合、またはステップ２２０５の終了後、変数ｗの値を１増加し（ステップ２２０６）、ステップ２２０２に戻る。そして、ステップ２２０２の比較において、変数ｗが並列度Ｗと同じであれば、処理を終了する（ステップ２２０２）。

【００５８】以上説明したパラレルスート、シリーズスート、単一ノードに対する実行時間の見積もり及び評価を、シリーズパラレルネスト木のルートノードから葉ノードへ再帰的に実行することにより、適切なハイパーブロックを生成することができる。すなわち、一つのハイパーブロックにまとめた方が実行時間が短くなるノードに関しては、一つのハイパーブロックにまとめられ、一つのハイパーブロックにまとめない方が実行時間が短くなるノードに関しては、別のハイパーブロックに分割する。

【００５９】次に、この第１の手法によるハイパーブロックの生成例を説明する。ここでは、並列処理を実行する方が実行時間が短くなる例と、条件分岐を実行した方が実行時間が短くなる例とを示すため、ハードウェアの並列度が６である場合と３である場合とを例として説明する。なお、図２乃至図６に示したプログラムを処理対象とし、各命令は時間１で実行されると仮定する。また、分岐処理によるペナルティ（分岐処理を行うことにより必然的に処理に要する実行時間）を５とする。

【００６０】まず、ハードウェアの並列度が６である場

合について説明する。例として、図６のシリーズパラレルネスト木において、スート５に関し、並列度Wが３である場合の実行時間の見積もりを考える。この場合、スート５は構成要素である基本ブロックＤ、Ｅが並列に接続されたパラレルスートである。したがって、図１４に示したパラレルスートに対する実行時間評価処理により、基本ブロックＤ、Ｅの並列度ごとの処理時間の最大値を求める。この場合、スート５の並列度Wが３であるから、基本ブロックＤ、Ｅの並列度は、基本ブロックＤの並列度が２、基本ブロックＥの並列度が１である場合と、基本ブロックＤの並列度が１、基本ブロックＥの並列度が２である場合とが考えられる。また、図３の基本ブロックＤを表す矩形の情報から、基本ブロックＤに関しては、クリティカルパス長が３であり、平均並列度が１（命令を並列に実行できず、逐次実行しなければならないことを意味する）である。したがって、基本ブロックＤの処理に要する時間は、並列度に関わらず３である。同様に、図３の基本ブロックＥを表す矩形の情報から、基本ブロックＥに関しては、クリティカルパス長が２であり、平均並列度が４である。したがって、基本ブロックＤの処理に要する時間は、並列度が１のときで８、並列度が２のときで４、並列度が３のときで３、並列度が４以上では２となる。したがって、上述した基本ブロックＤの並列度が２、基本ブロックＥの並列度が１である場合は、基本ブロックＤの実行時間が３、基本ブロックＥの実行時間が８である。一方、基本ブロックＤの並列度が１、基本ブロックＥの並列度が２である場合は、基本ブロックＤの実行時間が３、基本ブロックＥの実行時間が４である。すなわち、基本ブロックＤ、Ｅを並列に実行するとすると、基本ブロックＤの並列度が２、基本ブロックＥの並列度が１である場合は実行時間が８となり、基本ブロックＤの並列度が１、基本ブロックＥの並列度が２である場合は実行時間が４となる。そこで、スート５の最短処理時間は４と見積もることができる。図２４は、以上のような最短処理時間の見積もりを、図６のシリーズパラレルネスト木における各ノード（スート）に対して、並列度１～６の各場合で求めた結果を示す図である。

【００６１】ここで、図６のシリーズパラレルネスト木におけるパラレルスートであるスート５及びスート２の実行時間見積処理においては、一つのハイパーブロックにまとめるか、別の独立したハイパーブロックを生成するかの判断を行う（図１３参照）。例として、スート２の場合を考えると、図２４を参照して、並列度６のときのスート２の最短処理時間は６となる（基本ブロックＢの並列度が２でスート３の並列度が４である場合、または基本ブロックＢ及びスート３の並列度が共に３である場合）。これに対し、図１６、１７に示す利得見積処理によれば、スート３と基本ブロックＢとを、条件分岐をそのまま実行した場合の処理時間は、並列度６のときの基本ブロックＢの最短処理時間が４、スート３の最短処理時間が５であるから、９．５（＝５＋（４＋５）／２）である。したがって、条件分岐を行う場合の処理時間は並列処理における最短処理時間よりも大きいため、利得は０よりも大きくなる（９．５－６＝３．５＞０）。これにより、スート２は一つのハイパーブロックにまとめられることとなる（図１３、ステップ１３０４、１３０６参照）。スート２は、他のノードである基本ブロックＡ、Ｇとシリーズスートを構成しているため、最大の並列度６の場合についてのみ考察すればよいが、スート５の場合は、スート２を構成するスート３にどれだけの並列度が割り当てられるかに応じて、それぞれ利得を見積もる必要がある。煩雑になるので記載は省略するが、同様の計算を各並列度におけるスート５に対して行うと、全ての場合で利得が０よりも大きくなる。したがって、ハードウェアの並列度が６の場合は、全ての基本ブロックを一つにまとめたハイパーブロックを生成することとなる。図２４において、アスタリスク（＊）の付されたスート（スート０）は、独立したハイパーブロックを構成するという意味の印である（図７におけるnode.hyper_block = TRUE）。

【００６２】次に、ハードウェアの並列度が３である場合について説明する。ハードウェアの並列度が６である場合について説明したのと同様の手法で、図６のシリーズパラレルネスト木の各ノードに対して並列度１～３の各場合で求めた最短処理時間の見積もりを図２５に示す。まず、スート２について考える。スート２を並列実行する場合、最短処理時間は１２となる（基本ブロックＢの並列度が１でスート３の並列度が２である場合）。これに対し、図１６、１７に示す利得見積処理によれば、スート３と基本ブロックＢとを、条件分岐をそのまま実行した場合の処理時間は、並列度３のときの基本ブロックＢの最短処理時間が４、スート３の最短処理時間が６であるから、１０（＝５＋（６＋４）／２）である。したがって、利得は－２（＝１０－１２）であり、０よりも小さいので、スート２は二つのハイパーブロックに分割される（図１３、ステップ１３０４、１３０５参照）。また、基本ブロックＢとスート３とでは、スート３の方が最短処理時間が大きいので、スート３を独立のハイパーブロックとし、基本ブロックＢは基本ブロックＡ及び基本ブロックＧとシリーズスートを構成するハイパーブロックに含める（図１８参照）。次に、スート５について考える。上述したようにスート３は、独立のハイパーブロックであり、当該スート３においてスート５は、基本ブロックＣおよび基本ブロックＦとシリーズスートを構成する。したがって、スート５の並列度はハードウェアの並列度３をそのまま適用できる。この場合、スート５の最短処理時間は４である（基本ブロックＤの並列度が１で基本ブロックＥの並列度が２である場合）。これに対し、図１６、１７に示す利得見積処理に

よれば、基本ブロックD、Eを、分岐条件をそのまま実行した場合の処理時間は、並列度3のときの基本ブロックD、Eとも最短処理時間が3であるから、8（＝5＋（3＋3）／2）である。したがって、利得は4（＝8－4）であり、0よりも大きいので、スート5は一つのハイパーブロックにまとめられる。以上の結果、基本ブロックA、B、Gが一つのハイパーブロックを形成し、基本ブロックC、D、E、Fが別の独立したハイパーブロックを形成することとなる。図25において、アスタリスク（*）の付されたスート（スート0、3）は、独立したハイパーブロックを構成するという意味の印である（図7におけるnode.hyper_block = TRUE）。

【0063】〔第2の手法〕次に、当該基本ブロック内の命令間における依存関係（Dependence Path）の情報を持たせる第2の手法について説明する。第2の手法では、プログラムの各基本ブロックに命令レベルでの依存関係に関する情報を持たせておく。そして、基本ブロックの実行時間の見積もりの際に、この依存関係に基づいて当該基本ブロックのクリティカルパス長を再計算する。

【0064】図26を参照して具体的に説明する。図26は、図3に示した基本ブロックC、D、E、Fの内部の命令における依存関係を説明する図である。図3に示したように、基本ブロックCのクリティカルパス長は1、基本ブロックDのクリティカルパス長は3、基本ブロックEのクリティカルパス長は2、基本ブロックFのクリティカルパス長は1である。したがって、第1の手法によれば、基本ブロックC、D、E、F（図5、6のスート3に対応）の最短処理時間は5以下にはなり得ない（図24のスート3の欄参照）。しかし、基本ブロックD、E、Fの内部の命令間の依存関係が図26に示すようになっていた場合、すなわち、基本ブロックDにおける命令②と基本ブロックFにおける命令　とに依存関係があり、基本ブロックDにおける命令　と基本ブロックFとの間には依存関係がない場合、基本ブロックFの命令　は基本ブロックDの命令　と並列に実行することが可能である。したがって、この命令レベルでの依存関係を考慮することにより、スート3の最短処理時間を4と見積もることができる。

【0065】以上の処理を実現するため、第2の手法では、基本ブロック・コードスケジューラ21において、基本ブロック内部の命令の依存関係に関する情報を取得する。基本ブロック・コードスケジューラ21は、まず、命令間の依存関係を示す依存DAG（Directed Acyclic Graph）の全てのパスを求める。そして、得られたパスをその長さの降順に並べ替えておく。図27は、依存DAGの全てのパスを、ノードへのポインタを張ることによって保持するイメージを表す図である。

【0066】次に、基本ブロックに付された情報に基づいて、ハイパーブロック生成部22がPDGを作成し、PDGをシリーズパラレルグラフに変換し、さらにシリーズパラレルグラフからシリーズパラレルネスト木を生成する行程は、第1の手法と同一である。したがって、ここでは詳細な説明を省略する。

【0067】次に、ハイパーブロック生成部22は、実行時間見積部23を用いて、シリーズパラレルネスト木における各ノードの実行時間を再帰的に見積もり、その結果に基づいてハイパーブロック選択処理を実行する（図10参照）。このハイパーブロック選択処理により、シリーズパラレルネスト木の各ノードに、独立したハイパーブロックとして扱うか否かを示す情報が付される。第2の手法におけるハイパーブロック選択処理は、基本的には第1の手法と同様の手順で実行されるが、基本ブロック内部の命令の依存関係を考慮し、必要に応じて当該基本ブロックのクリティカルパス長を変更する処理が実行される。以下、ハイパーブロック選択処理について、詳細に説明する。

【0068】ハイパーブロック選択処理の全体的な動作の流れは、図11を参照して説明した第1の手法と同様である。すなわち、まず、シリーズパラレルネスト木のノードの一つを処理対象とし（ステップ1101）、当該ノードの属性を調べ、パラレルスートか、シリーズスートか、基本ブロック（単一ノード）かを判断する（ステップ1102）。そして、判断結果に応じて、パラレルスートの実行時間見積処理（ステップ1103）、シリーズスートの実行時間を評価するための処理（ステップ1104、1105）、単一ノードの実行時間評価処理を行う（ステップ1106）。

【0069】また、パラレルスートの実行時間見積処理（ステップ1103）も、実行時間評価処理の内容を除き、全体的な動作の流れは図13を参照して説明した第1の手法と同様である。すなわち、まず、処理対象であるノードの子ノードに対して当該ハイパーブロック選択処理を再帰的に適用する（ステップ1301）。そして、パラレルスートの実行時間評価処理（ステップ1302）及び利得見積処理（ステップ1303）を行った後、得られた利得の値が0よりも大きいか否かを判断する（ステップ1304）。利得が0よりも大きい場合は、ノード選択処理を行う（ステップ1305）。利得が0以下である場合は、子ノード1と子ノード2の面積（すなわち、node.all：当該ノードの延べ実行時間）の和を親ノード（すなわち、初めに処理対象としたノード）の面積とする（ステップ1306）。

【0070】図28は、第2の手法において、図13におけるステップ1302の実行時間評価処理の内容を説明するフローチャートである。また、図29は、図28の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図28を参照すると、まず、この処理の中でのみ有効な自由変数w1の値を1に初期化する（ステップ2801）。また、この処理の中でのみ有効な自

由変数ｍｉｎ及びｍｉｎ２の値を無限大に初期化する（ステップ２８０２）。そして、変数ｗ１とハードウェアの並列度Wとを比較し、同じかどうか調べる（ステップ２８０３）。変数ｗ１が並列度Wと同じでなければ、次に、この処理の中でのみ有効な自由変数ｗ２を０に初期化し（ステップ２８０４）、この変数ｗ２と変数ｗ１とを比較して同じかどうか調べる（ステップ２８０５）。変数ｗ１と変数ｗ２とが同じでない場合、次に、子ノード１が並列度ｗ２のときの最短処理時間と、子ノード２が並列度（ｗ１－ｗ２）のときの最短処理時間とを比較し、大きい方の値を求める（ステップ２８０６）。そして、得られた値と、変数ｍｉｎの値とを比較し、小さい方の値を変数ｍｉｎの新たな値とする（ステップ２８０７）。また、子ノード１が並列度ｗ２のクリティカルパス長を無視した場合の処理時間と、子ノード２が並列度（ｗ１－ｗ２）のときのクリティカルパス長を無視した場合の処理時間とを比較し、大きい方の値を求める（ステップ２８０８）、そして、得られた値と、変数ｍｉｎ２の値とを比較し、小さい方の値を新たなｍｉｎ２の値とする（ステップ２８０９）。この後、変数ｗ２の値を１増加し（ステップ２８１０）、ステップ２８０５の比較に戻る。一方、変数ｗ１と変数ｗ２とが同じである場合は、現時点での変数ｍｉｎの値を親ノード（すなわち、初めに処理対象としたノード）が並列度ｗ１のときの最短処理時間とし、現時点での変数ｍｉｎ２の値を当該親ノードのクリティカルパス長を無視した場合の処理時間とする（ステップ２８１１）。そして、変数ｗ１の値を１増加し（ステップ２８１２）、ステップ２８０２に戻る。そして、ステップ２８０３の比較において、変数ｗ１と並列度Wとが同じであれば、依存パス融合処理（ステップ２８１３）を行った後に実行時間評価処理を終了する。

【００７１】図３０は、図２８におけるステップ２８１３の依存パス融合処理の内容を説明するフローチャートである。なお、依存パスとは、依存ＤＡＧにおいて命令間の依存関係を表すパスである。また、図３０において、node.n_exec_pathは、ノードにおける依存パスの数を示す。図３０を参照すると、まず、親ノードにおける依存パスの数を子ノード１における依存パスの数と子ノード２における依存パスの数の和とする（ステップ３００１）。次に、この処理の中でのみ有効な自由変数ｎ１、ｎ２、ｎを０に初期化する（ステップ３００２）。そして、変数ｎと親ノードの依存パスの数とを比較し、同じかどうか調べる（ステップ３００３）。変数ｎと親ノードの依存パスの数とが同じでない場合、次に、子ノード１のｎ１番目の依存パスの長さと、子ノード２のｎ２番目の依存パスとの長さを比較する（ステップ３００４）。そして、子ノード１のｎ１番目の依存パスの方が長い場合は、親ノードのｎ番目の依存パスを、子ノード１のｎ１番目の依存パスとし、変数ｎ１の値を１増加する（ステップ３００５）。また、子ノード２のｎ２番目の依存パスの方が長い場合は、親ノードのｎ番目の依存パスを、子ノード２のｎ２番目の依存パスとし、変数ｎ２の値を１増加する（ステップ３００６）。ステップ３００５またはステップ３００６の後、変数ｎの値を１増加し（ステップ３００７）、ステップ３００３へ戻る。そして、変数ｎと親ノードの依存パスの数とが同じならば、依存パス融合処理を終了する（ステップ３００３）。

【００７２】以上の動作により、並列度Wのときのパラレルスートの最短処理時間の見積もりが得られる。上記のように、第２の手法では、クリティカルパス長を無視した（すなわち、実行時間がクリティカルパス長を下回ることを許す）場合の処理時間の見積もり（図７のnode.best_time2）も計算される。この値は、後述するシリーズスートの実行時間の評価において用いられる。また、依存パス融合処理において、依存パスの融合とソートが行われる。子ノード１、２の依存パスは、基本ブロック・コードスケジューラ２１により長いものから降順にソートしてあるので、これを用いて親ノードにおける依存パスが再構成される。

【００７３】図３１は、第２の手法において、図１１におけるステップ１１０５のシリーズスートに対する実行時間評価処理の内容を説明するフローチャートである。また、図３２は、図３１の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図３１を参照すると、まず、処理対象であるノードの延べ実行時間を、子ノード１、２の延べ実行時間の和とする（ステップ３１０１）。次に、クリティカルパス長再計算処理（ステップ３１０２）、最短処理時間計算処理（ステップ３１０３）を順次実行する。

【００７４】図３３は、図３１におけるステップ３１０２のクリティカルパス長再計算処理内容を説明するフローチャートである。図３３を参照すると、まず、この処理の中でのみ有効な自由変数ｎ１、ｉｄｘを０に初期化する（ステップ３３０１）。そして、変数ｎ１の値と子ノード１の依存パスの数とを比較し、同じかどうかを調べる（ステップ３３０２）。変数ｎ１の値と子ノード１の依存パスの数とが同じでない場合、次に、この処理の中でのみ有効な自由変数ｎ２を０に初期化する（ステップ３３０３）。そして、変数ｎ２の値と子ノード２の依存パスの数とを比較し、同じかどうかを調べる（ステップ３３０４）。変数ｎ２の値と子ノード２の依存パスの数とが同じでない場合、次に、子ノード１のｎ１番目の依存パスの最後のノードが、子ノード２のｎ２番目の依存パスの最初のノードに依存しているかどうか調べる（ステップ３３０５）。そして、依存しているならば、当該子ノード１のｎ１番目の依存パスと子ノード２のｎ２番目の依存パスとを結合し、親ノード（すなわち、初めに処理対象としたノード）のｉｄｘ番目の依存パスと

する（ステップ３３０６）。ステップ３３０６の処理の後、及びステップ３３０５において、子ノード１のｎ１番目の依存パスの最後のノードが、子ノード２のｎ２番目の依存パスの最初のノードに依存していない場合、変数ｗ２の値を１増加し（ステップ３３０７）、ステップ３３０４に戻る。一方、ステップ３３０４において、変数ｎ２の値と子ノード２の依存パスの数とが同じならば、変数ｎ１の値を１増加し（ステップ３３０８）、ステップ３３０２へ戻る。そして、ステップ３３０２において、変数ｎ１の値と子ノード１の依存パスの数とが同じならば、処理を終了する。

【００７５】以上のようにして、所定のシリーズスートの子ノードにおいて、その子ノード中の命令の依存パスを連結し、この連結された依存パスの長さに基づいてソートした上で、当該シリーズスートのクリティカルパス長を再計算する。これにより、子ノード中の命令の依存関係によっては、当該シリーズスート全体におけるクリティカルパス長を、子ノードのクリティカルパス長を単純に足した場合よりも短くすることができる。

【００７６】図３４は、図３１におけるステップ３１０３の最短処理時間計算処理の内容を説明するフローチャートである。図３４を参照すると、まず、親ノードの依存パスを長さの降順にソートする（ステップ３４０１）。次に、この処理の中でのみ有効な自由変数ｗを０に初期化する（ステップ３４０２）。そして、変数ｗとハードウェアの並列度Ｗとを比較し、同じかどうかを調べる（ステップ３４０３）。変数ｗと並列度Ｗとが同じでない場合、クリティカルパス長を考慮しない場合における親ノードの実行時間を、子ノード１、２のクリティカルパス長を考慮しない場合における実行時間の和とする（ステップ３４０４）。そして、変数ｗの値を１増加し（ステップ３４０５）、ステップ３４０３へ戻る。ステップ３４０３において、変数ｗと並列度Ｗとが同じであれば、処理を終了する。

【００７７】以上のようにして、子ノード１、２におけるクリティカルパス長を考慮しない場合における最短処理時間の和において、再計算された当該シリーズスートのクリティカルパス長を下回らない値を当該シリーズスートの最短処理時間と見積もることができる。

【００７８】図３５は、第２の手法において、図１１におけるステップ１１０６の単一ノードに対する実行時間評価処理の内容を説明するフローチャートである。また、図３６は、図３５の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図３５を参照すると、まず、この処理の中でのみ有効な自由変数ｗを１に初期化する（ステップ３５０１）。そして、変数ｗとハードウェアの並列度Ｗとを比較し、同じかどうか調べる（ステップ３５０２）。変数ｗが並列度Ｗと同じでなければ、次に、処理対象であるノードの最短処理時間を、基本ブロック中の延べ実行時間を変数ｗで割った値とする（ステップ３５０３）。また、処理対象であるノードのクリティカルパスを考慮しない場合における実行時間も、基本ブロック中の延べ実行時間を変数ｗで割った値とする（ステップ３５０４）。そして、ステップ３５０３で得られた値が基本ブロックのクリティカルパス長を下回ったかどうかを判断する（ステップ３５０５）。ステップ３５０３で算出された処理対象であるノードの最短処理時間が、基本ブロックのクリティカルパス長を下回っている場合、当該ノードの最短処理時間を基本ブロックのクリティカルパス長とする（ステップ３５０６）。ステップ３５０３で算出された処理対象であるノードの最短処理時間が、基本ブロックのクリティカルパス長を下回っていない場合、またはステップ３５０６の終了後、変数ｗの値を１増加し（ステップ３５０７）、ステップ３５０２に戻る。そして、ステップ３５０２の比較において、変数ｗが並列度Ｗと同じであれば、処理を終了する（ステップ３５０２）。

【００７９】以上、説明した第２の手法は、基本ブロック・コードスケジューラ２１において依存ＤＡＧのパス（依存パス）のソートを行う。この処理は、依存パスの数をｍとした場合、ｍ×log ｍに比例する計算時間を要する。また、依存パスの連結には最長でm^2に比例する計算時間を要し、連結された依存パスのソートにやはりｍ×log ｍに比例する計算時間を要する。したがって、上述した処理は、最長でｎ×m^2に比例する計算時間を要することとなる。しかしながら、プログラム中の最適化しようとする部分が多くの基本ブロックに分割されている場合、依存パスの数ｍは小さな値となる。したがって、実践的には大きな計算時間を要しないで実行が可能である。

【００８０】次に、第２の手法による最短処理時間の計算例を説明する。図３７は、図２、３のプログラムに対して第２の手法により得られた最短処理時間の見積もりを、図６のシリーズパラレルネスト木における各ノード（スート）に対して、並列度１～６の各場合で求めた結果を示す図である。また、図３８は、基本ブロックのクリティカルパス長を無視した場合における最短処理時間の見積もりを、同様にして求めた結果を示す図である。ここで、基本ブロックＤ、Ｅ、Ｆの内部における命令の依存関係は、図２６に示したようになっている。

【００８１】したがって、第１の手段により得られた最短処理時間の見積もりを示す図２４と、図３７とを比較すると、基本ブロックＤ、Ｅ、Ｆの命令の依存関係が、スート３における見積もりの結果に現れている。スート３は、基本ブロックＦとスート４とで構成されたシリーズスートである。ここで、スート３における並列度６の場合の最短処理時間は図３７によれば４であり、図２４の場合における５よりも１だけ少なくなっている。そして、この値は、図３８における該当個所の最短処理時間が４であることから、再計算されたクリティカルパス長

を下回らないことがわかる。したがって、この場合のスート３の最短処理時間は４と見積もられ、第１の手法の場合と比べて最適化が進んでいる。

【００８２】

【発明の効果】以上説明したように、本発明によれば、プログラムの所定の領域に対して高速かつ適切なハイパーブロックの生成を行うことができるため、最適化処理において、ある程度実行可能性が高い多くのパスにおける実行効率を向上させることができる。

【図面の簡単な説明】

【図１】　本発明の実施の形態におけるコンパイラの構成を説明する図である。

【図２】　処理対象であるプログラムの最適化処理を行う領域の制御フローグラフと当該部分の命令列のリストを示す図である。

【図３】　図２に示したプログラム領域において、node.clを縦の長さとし、node.allを面積とする矩形領域で基本ブロックを表した図である。

【図４】　図３から作成されたＰＤＧを示す図である。

【図５】　図４のＰＤＧから変換されたシリーズパラレルグラフを示す図である。

【図６】　図５のシリーズパラレルグラフから生成されたシリーズパラレルネスト木を示す図である。

【図７】　本実施の形態の動作説明に用いる記号を定義した図表である。

【図８】　ＰＤＧをシリーズパラレルグラフに変換するアルゴリズムを示す擬似プログラムを示す図である。

【図９】　シリーズパラレルグラフからシリーズパラレルネスト木を生成するアルゴリズムを示す疑似プログラムを示す図である。

【図１０】　本実施の形態におけるハイパーブロック生成部の全体動作を説明するフローチャートである。

【図１１】　ハイパーブロック選択処理の全体的な動作の流れを示すフローチャートである。

【図１２】　図１１に対応する動作のアルゴリズムを示す疑似プログラムを示す図である。

【図１３】　パラレルスートに対する実行時間見積処理の内容を説明するフローチャートである。

【図１４】　実行時間評価処理の内容を説明するフローチャートである。

【図１５】　図１４の実行時間評価処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図１６】　利得見積処理の内容を説明するフローチャートである。

【図１７】　図１６の利得見積処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図１８】　ノード選択処理の内容を説明するフローチャートである。

【図１９】　図１８のノード選択処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図２０】　シリーズスートに対する実行時間評価処理の内容を説明するフローチャートである。

【図２１】　図２０の実行時間評価処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図２２】　単一ノードに対する実行時間評価処理の内容を説明するフローチャートである。

【図２３】　図２２の実行時間評価処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図２４】　最短処理時間の見積もりを、図６のシリーズパラレルネスト木における各ノード（スート）に対して、並列度１～６の各場合で求めた結果を示す図である。

【図２５】　最短処理時間の見積もりを、図６のシリーズパラレルネスト木における各ノード（スート）に対して、並列度１～３の各場合で求めた結果を示す図である。

【図２６】　図３に示した基本ブロックＣ、Ｄ、Ｅ、Ｆの内部の命令における依存関係を説明する図である。

【図２７】　依存ＤＡＧの全てのパスを、ノードへのポインタを張ることによって保持するイメージを表す図である。

【図２８】　第２の手法において、実行時間評価処理の内容を説明するフローチャートである。

【図２９】　図２８の実行時間評価処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図３０】　依存パス融合処理の内容を説明するフローチャートを示す図である。

【図３１】　第２の手法において、シリーズスートに対する実行時間評価処理の内容を説明するフローチャートである。

【図３２】　図３１の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。

【図３３】　クリティカルパス長再計算処理内容を説明するフローチャートである。

【図３４】　最短処理時間計算処理の内容を説明するフローチャートである。

【図３５】　第２の手法において、単一ノードに対する実行時間評価処理の内容を説明するフローチャートである。

【図３６】　図３５の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。

【図３７】　第２の手法により得られた最短処理時間の見積もりを、図６のシリーズパラレルネスト木における各ノード（スート）に対して、並列度１～６の各場合で求めた結果を示す図である。

【図３８】　基本ブロックのクリティカルパス長を無視した場合における最短処理時間の見積もりを、同様にして求めた結果を示す図である。

【図３９】　処理対象のプログラムの構成を基本ブロックで表現した例を示す図である。

【図４０】　パラレルスートの実行時間を構成要素のパラメータ（ｄ、ｗ）で表現できることを説明する図である。

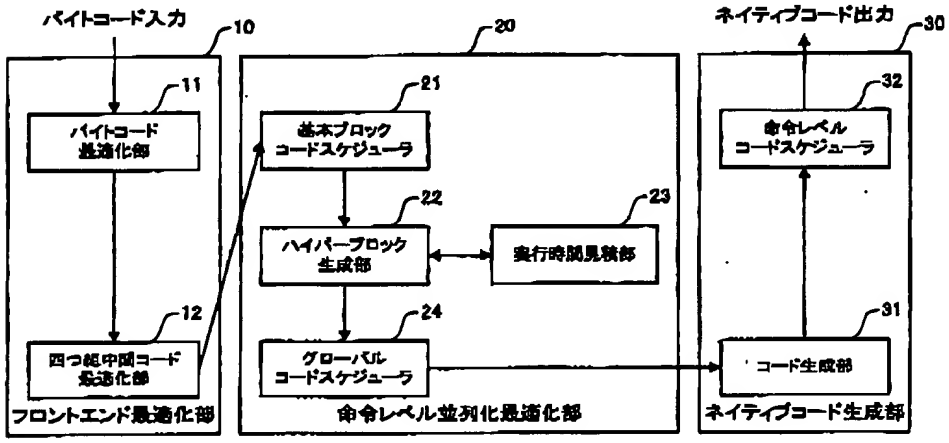
【図４１】　モデル化された基本ブロックによるパラレルスートの例を示す図である。

【図４２】　基本ブロック内部における隙間を説明する図である。

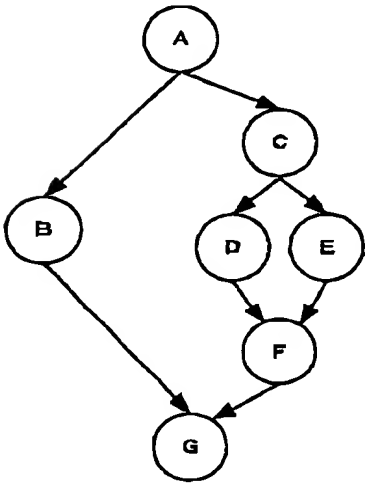
【符号の説明】

１０…フロントエンド最適化部、１１…バイトコード最適化部、１２…四つ組中間コード最適化部、２０…命令レベル並列化最適化部、２１…基本ブロック・コードスケジューラ、２２…ハイパーブロック生成部、２３…実行時間見積部、２４…グローバル・コードスケジューラ、３０…ネイティブコード生成部、３１…コード生成部、３２…命令レベル・コードスケジューラ

【図１】



【図２】

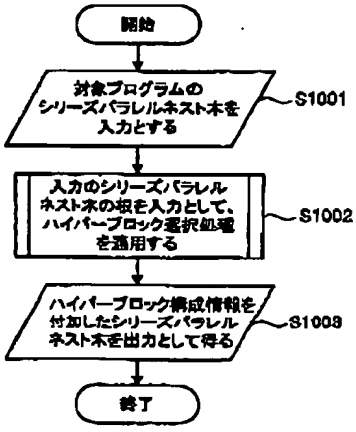


制御フローグラフ

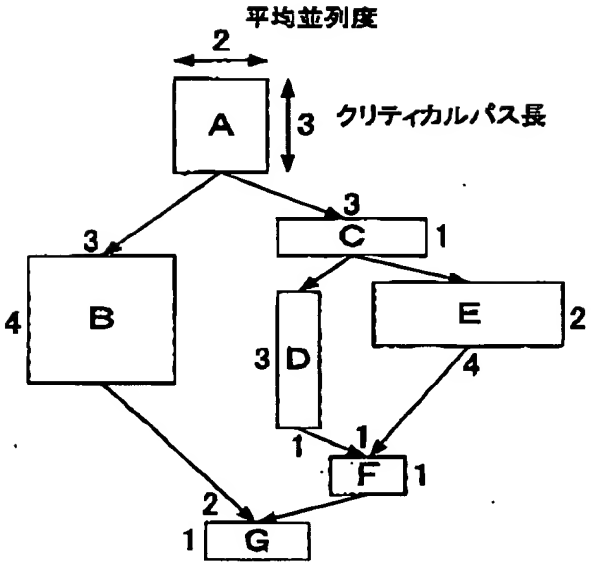
```
A:imove a, 1
  imove b, 1
  imove c, 3
  iadd  a, 1
  isub  b, 1
    cmpjmp-eq a, 1, C
B:iload  d, [base+a]
  iload  e, [base+b]
  iload  f, [base+c]
  iadd  d, 1
  iadd  e, 1
  iadd  f, 1
  isub  d, 1
  isub  e, 2
  isub  f, 3
  iadd  d, e
  iadd  e, f
  goto G:
```

```
C:imove d, 4
  imove e, 4
    cmpjmp-eq b, 1, E
D:iload  g, [base+a]
  iadd  g, 1
  .istore [base+a], g
  goto F:
E:iload  h, [base2]
  iload  i, [base2+a]
  iload  j, [base2+b]
  iload  k,[base2+c]
  iadd  h, 1
  iadd  i, 2
  iadd  j, 3
  iadd  k, 4
F: iadd j, g
G:iadd e, i
  iadd f, d
```

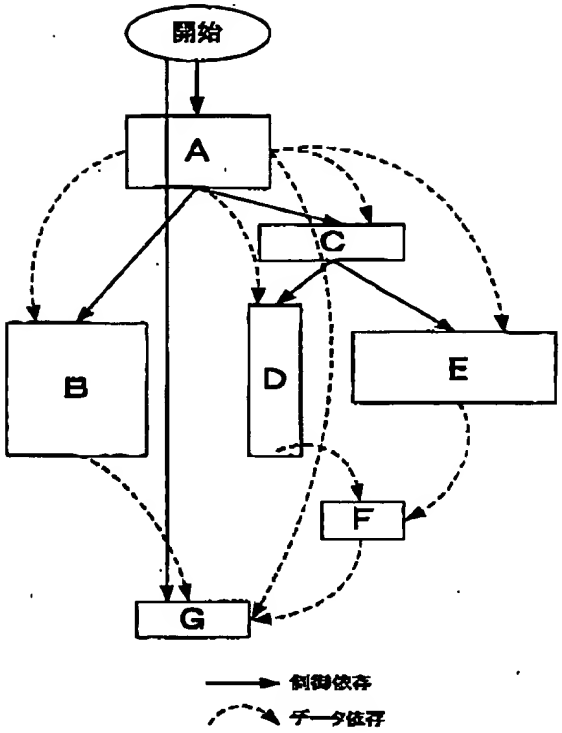
【図１０】



【図３】



【図４】

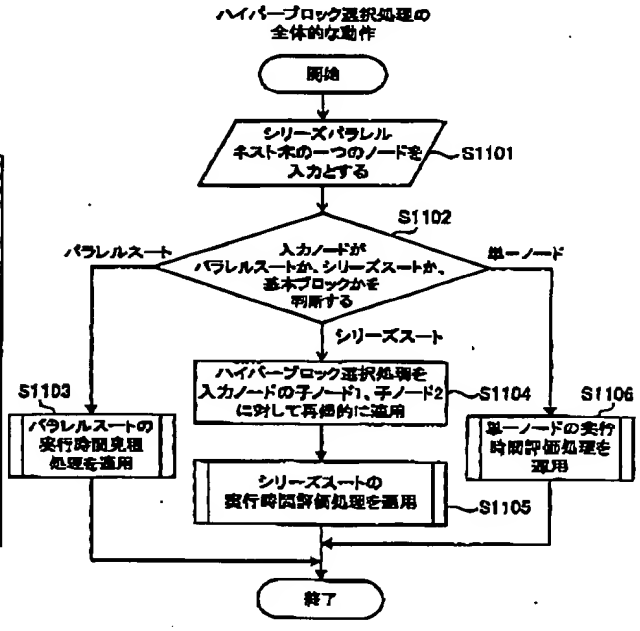


【図８】

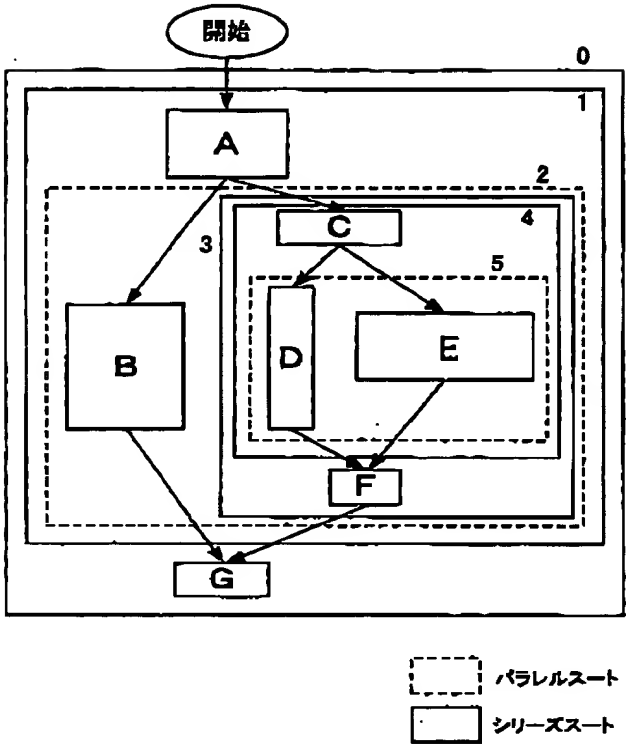
PDGをシリーズパラレルグラフに変換する擬似プログラム

```
Input: PDG, dominator tree of the PDG
Output: transformed PDG
for n in all the PDG node
begin
  if(the number of predecessor of n is more than 1)
  begin
    for all the combination(p1, p2) of the predecessor of n
    begin
      if(p1 dominates p2)
        remove an edge from p1 to n
      if(p2 dominates p1)
        remove an edge from p2 to n
    end
  end
end
```

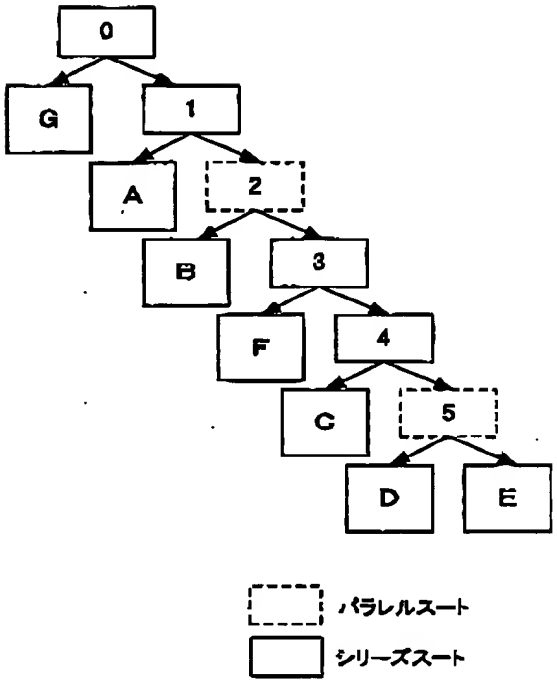
【図１１】



【図5】



【図6】

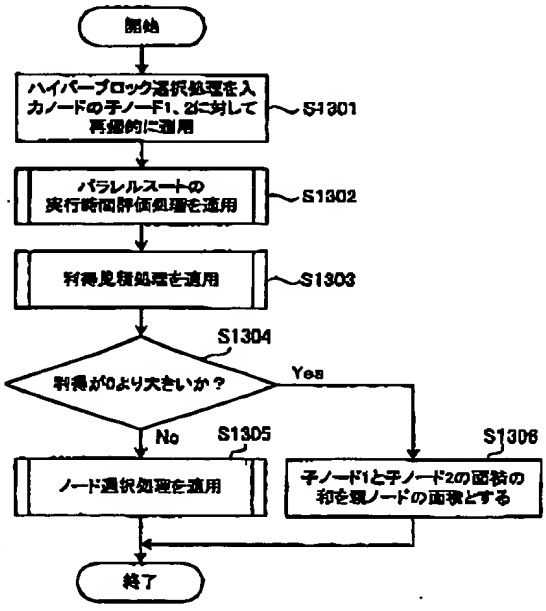


【図13】

【図7】

W	ハードウェアの並列度(定数)。
w, w1, w2	主にハードウェアの並列度を表す。ループの制御変数として使われる。それぞれの出現で局所的。
node	シリーズパラレルネスト木の一つのノードを表す。
node.attributes	ノードが基本ブロック、シリーズスート、パラレルスートのいずれであるかを表す。
node.cl	ノードが表すスートのクリティカルパス長(ノードが基本ブロックである場合にのみ定義される)。
node.s3	ノードの積べ実行時間。
node.child1 node.child2	ノードのネストの内側のシリーズスート、パラレルスート、基本ブロックを示す(ノードがシリーズスートまたはパラレルスートである場合にのみ定義される)。
node.dep_path[n] node.dep_path.in node.dep_path.out node.dep_path.length node.n_dep_path	ノードが示す領域内の依存パスのリスト。ハイパーブロック選択処理の第2の手法で、基本ブロックのコードスケジューリングを行う際に生成する。dep_path.lengthは、依存パスのヘッド位置にあるDAGノードへのポインタ。dep_path.outは、依存パスのテール位置にあるDAGノードへのポインタ。
node.best_time[w]	ノードが表すスートの最短処理時間の見積もり。定義域は1≦w≦W。
node.best_time2[w]	クリティカルパス長を無視した場合のノードの処理時間の見積もり。実行時間がクリティカルパス長以下になる場合を許す。ハイパーブロック選択処理の第2の手法で見積もられ、シリーズスートの実行時間の見積もりに用いられる。定義域は1≦w≦W。
node.hyper_block	旗としてTRUEかFALSEかをとる。独立のハイパーブロックを生成する場合、TRUEになる。

パラレルスートの実行時間見積処理



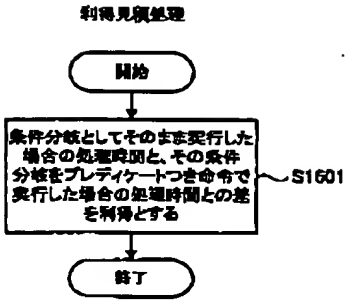
【図９】

シリーズパラレルネスト木を生成する擬似プログラム

```
Input: Series-Parallel Graph
Output: suite nest tree

apply following "algorithm" to the header node of Series-Parallel Graph
algorithm(node)
begin
    allocate new root R of suite nest tree
    if(node has more than 1 successor)
    begin
        for each successor s(i) of node
        begin
            put (recursively called)algorithm(s(i)) as the leaf of R
        end
        R.attribute = parallel suite;
        return R;
    end
    else
    begin
        put (recursively called)algorithm(the successor of node) as the leaf of R
        R.attribute = series suite;
        return R;
    end
end
```

【図１６】



【図１２】

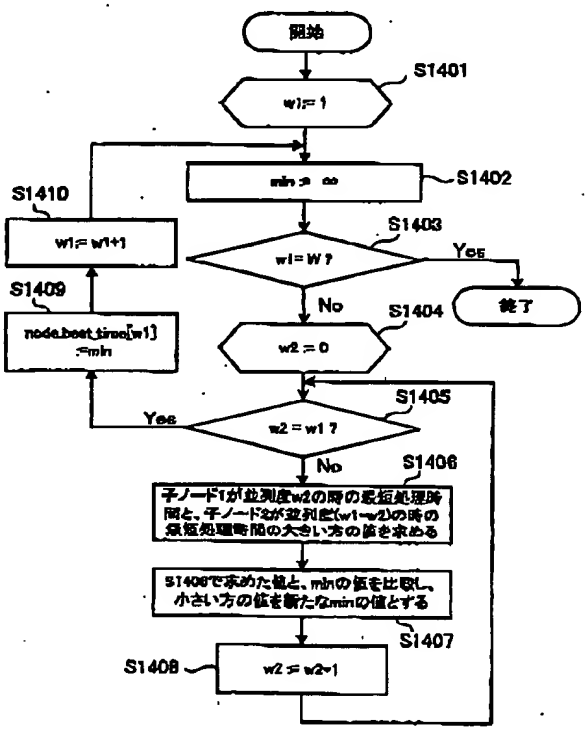
ハイパーブロック選択処理の擬似プログラム

```
Hyper_Block_Selector(Node node)
begin
    switch(node.attribute)
    begin
    case: single node
        ************************************
        single node
        ************************************
        single node_evaluator(node);
        return;
    break;
    case: Series-Suite
        ************************************
        Series Suite
        ************************************
        Hyper_Block_Selector(node.child1);
        Hyper_Block_Selector(node.child2);
        series_suite_evaluator(node.child1,node.child2)
        break;
    case: Parallel-Suite
        ************************************
        Parallel Suite
        ************************************
        Hyper_Block_Selector(node.child1);
        Hyper_Block_Selector(node.child2);
        // do parallel schedule
        parallel_suite_evaluator(node.child1,node.child2);

        // whether hyperblock is formed or not
        gain = gain_estimator(node.child1, node.child2, node);
        if( gain <= 0 )
        begin
            //choose one with some heuristics
            //the other is flaged as separate hyper_block
            node_selector(node.child1, node.child2);
        end
        else
            node.all = node.child1.all + node.child2.all;
        break;
    end of switch
end
```

【図１４】

パラレルスートの実行時間評価処理



【図１５】

パラレルスートの実行時間評価処理の擬似プログラム

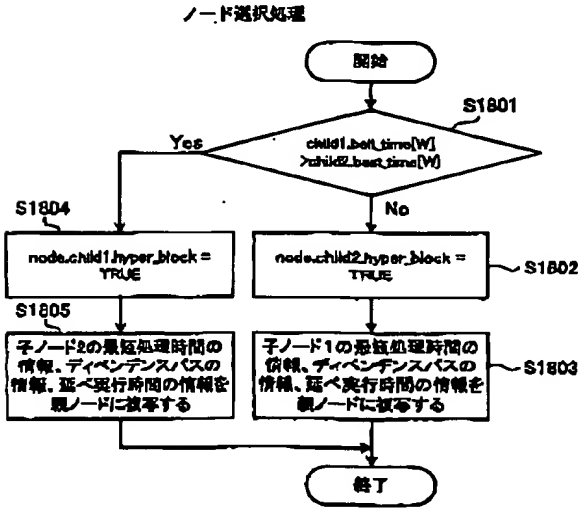
```
procedure parallel_suite_evaluator(node, child1, child2)
begin
int min, w1, w2;
for w1=1 to W
begin
        min = infinite;
        for w2=1 to w1
        begin
            min = min( min, max(child1.best_time[w2], child2.best_time[w1-w2] );
        end
        node.best_time[w1] = min;
end
end
```

【図１７】

利得見積処理の擬似プログラム

```
procedure gain_estimator(node)
    return branch_penalty+(node.child1.best_time[W]+node.child2.best_time[W])/2
                - node.best_time[W] ;
```

【図１８】



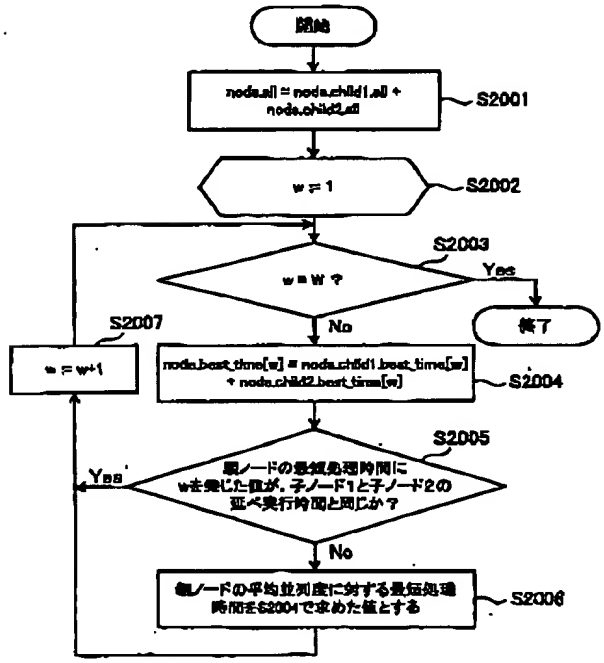
【図１９】

ノード選択処理の擬似プログラム

```
procedure node_selector(child1, child2)
begin
    if(child1.best_time[W] > child2.best_time[W])
    begin
        node.child2.hyper_block=TRUE;
        copy node.child1.best_time to node.best_time;
        copy node.child1.dep_path to node.dep_path;
    end
    else
    begin
        node.child1.hyper_block=TRUE;
        copy node.child2.best_time to node.best_time;
        copy node.child2.dep_path to node.dep_path;
    end
end
```

【図２０】

シリーズスートの実行時間評価処理



【図２１】

シリーズスートの実行時間評価処理の擬似プログラム

```
procedure series_suite_evaluator(node, child1, child2)
begin
int w;
node.all = node.child1.all + node.child2.all;
for w=1 to W
begin
   node.best_time[w] = node.child1.best_time[w] + node.child2.best_time[w];
   if(node.best_time[w] * w != node.child1.all+ node.child2.all)
         node.best_time[
               (node.child1.all+node.child2.all)/node.best_time[w]
         ] = node.best_time[w];
end
```

【図２３】

単一ノードの実行時間評価処理の擬似プログラム

```
procedure singlenode_evaluator(node, w)
    return max(node.all/w, node.cl);
```

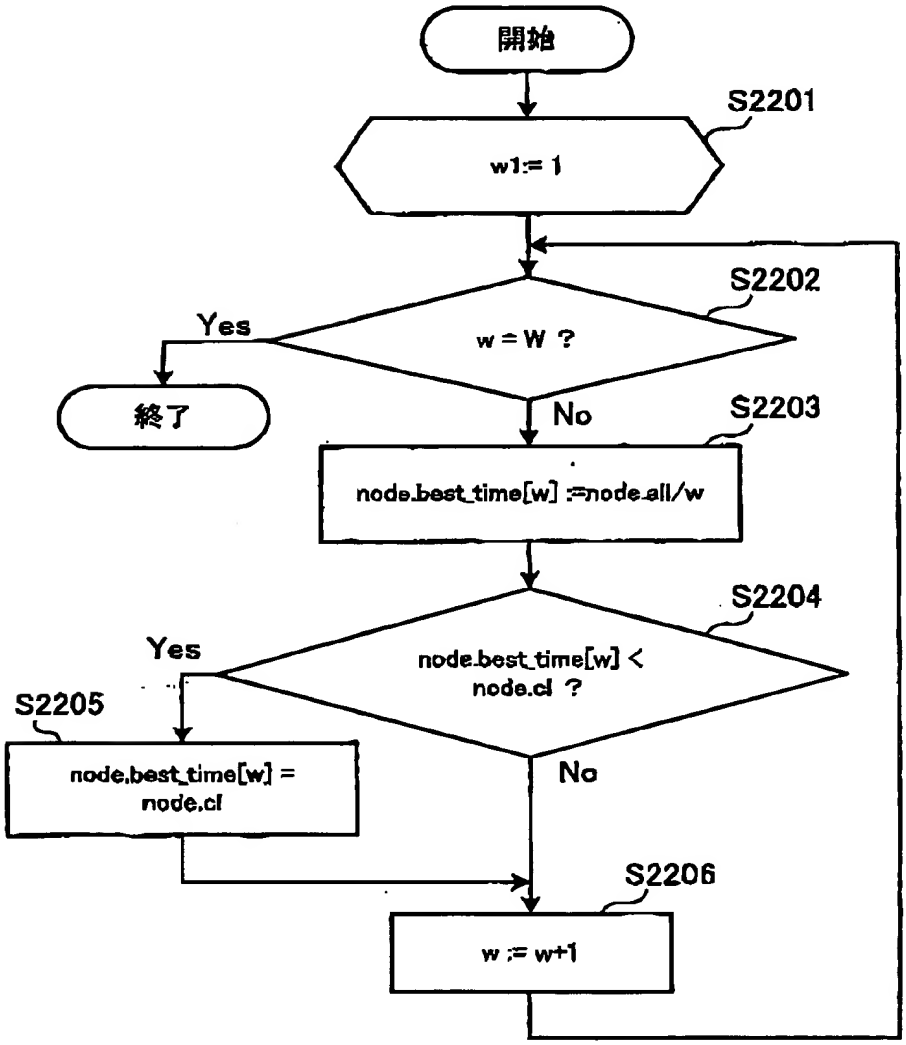
【図３６】

単一ノードの実行時間評価処理の擬似プログラム

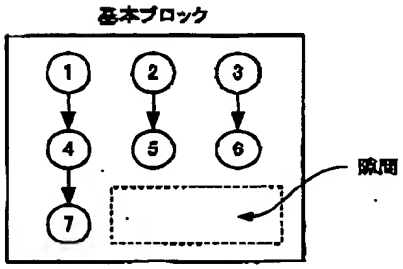
```
procedure singlenode_evaluator(node)
begin
int w;
for w = 1 to W
begin
    node.best_time2[w] = node.all/w;
    node.best_time[w] = max(node.all/w, node.cl);
end
end
```

【図２２】

単一ノードの実行時間評価処理



【図４２】



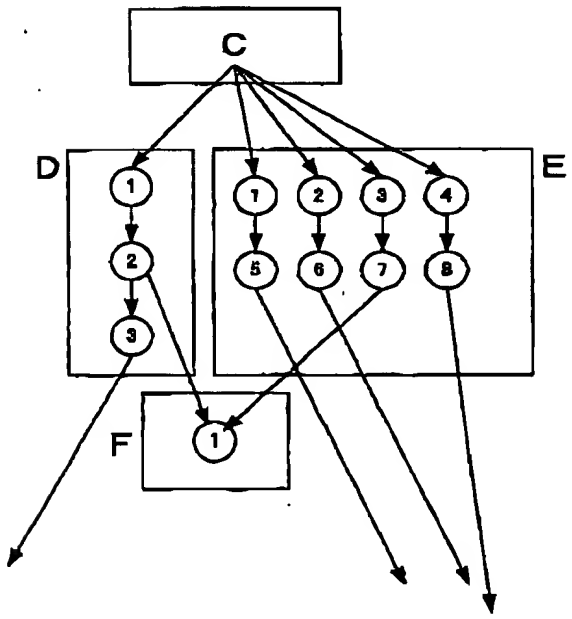
【図24】

スート/並列度	1	2	3	4	5	6
A	6	3	3	3	3	3
B	12	6	4	4	4	4
C	3	2	1	1	1	1
D	3	3	3	3	3	3
E	8	4	3	2	2	2
F	1	1	1	1	1	1
G	2	1	1	1	1	1
0(*)	35	19	16	15	10	10
1	33	18	15	14	9	9
2	27	15	12	11	6	6
3	15	11	6	6	5	5
4	14	10	5	4	4	4
5	11	8	4	3	3	3

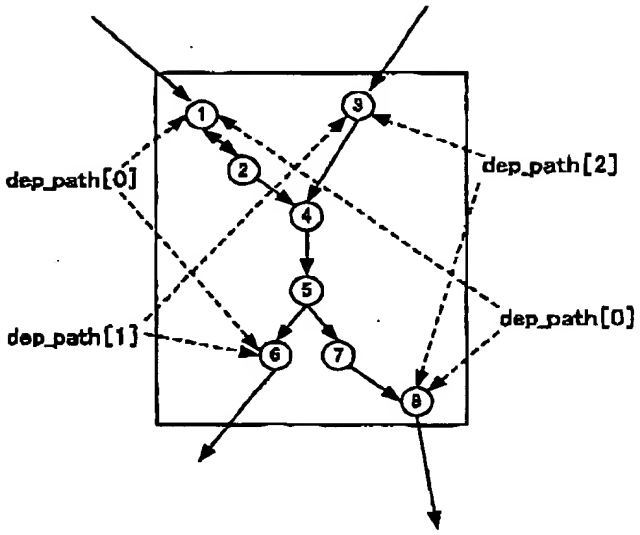
【図25】

スート/並列度	1	2	3
A	6	3	3
B	12	6	4
C	3	2	1
D	3	3	3
E	8	4	3
F	1	1	1
G	2	1	1
0(*)	20	10	8
1	18	9	7
2	12	6	4
3(*)	16	11	6
4	15	10	5
5	12	8	4

【図26】

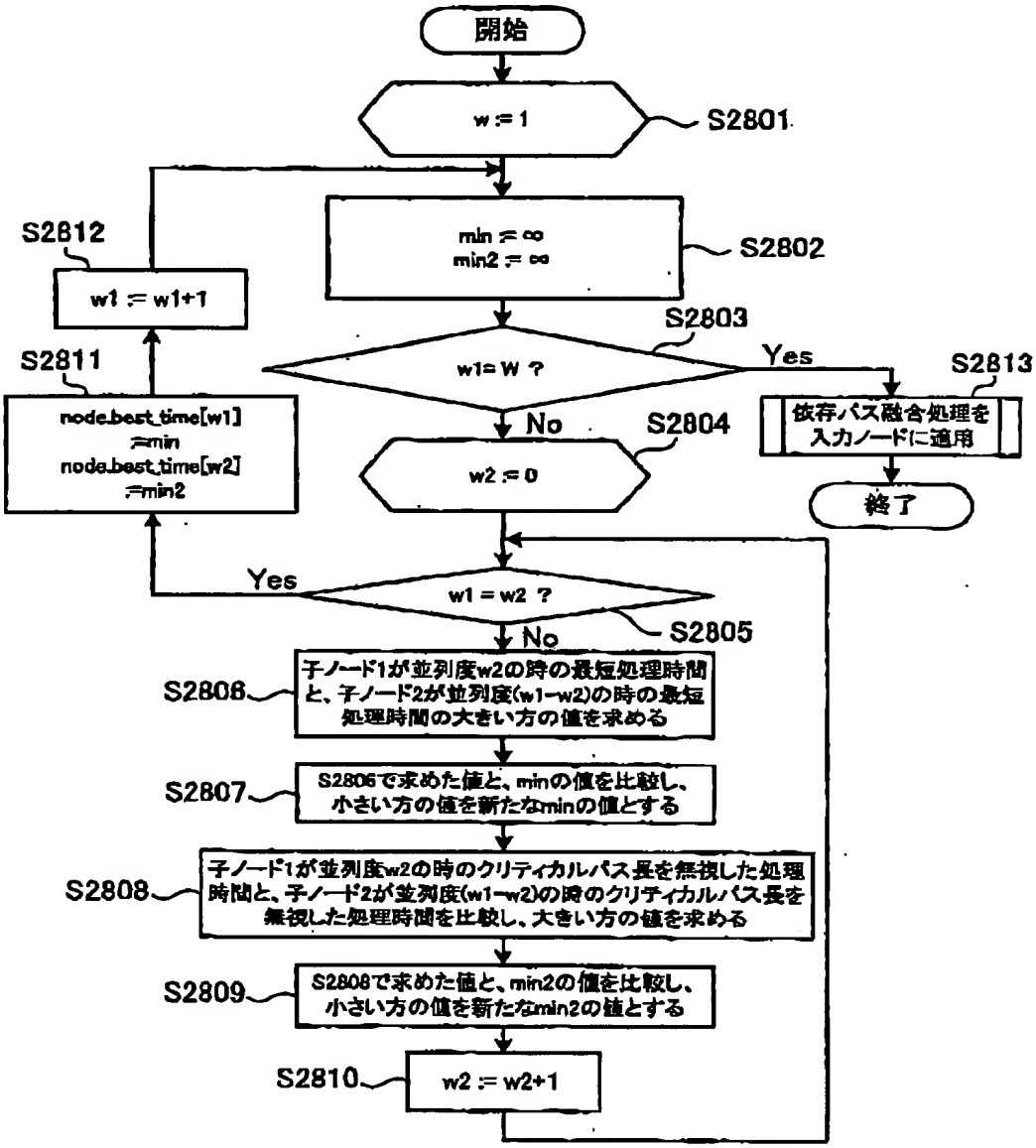


【図27】



【図２８】

パラレルスート実行時間評価処理



【図２９】

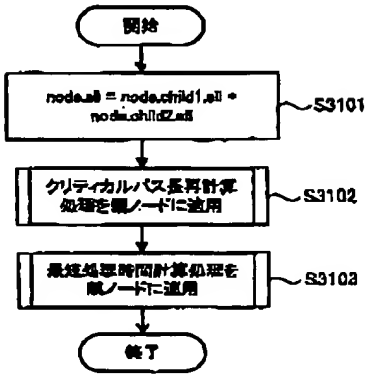
パラレルスートの実行時間評価処理の擬似プログラム

```
procedure parallel_suite_evaluation(node, child1, child2)
begin
int min = infinite;
int min2 = infinite;
int w1, w2, n, n1, n2;
for w1=1 to W
begin
  for w2=1 to w1
  begin
    min = min( min, max(child1.best_time[w2], child2.best_time[w1-w2] );
    min2 = min( min2, max(child1.best_time2[w2], child2.best_time2[w1-w2] );
  end
  node.best_time[w1] = min;
  node.best_time2[w1] = min2;
end

// merge child1.dep_path & child2.dep_path into node.dep_path
n1 = 0;     n2 = 0;
for n = 0 to node.n_dep_path - 1
    if(child1.dep_path[n1].length > child2.dep_path[n2].length)
      node.dep_path[n] = child1.dep_path[n1++];
    else
      node.dep_path[n] = child2.dep_path[n2++];
end
node.n_dep_path = child1.n_dep_path + child2.n_dep_path;
end
```

【図３１】

シリーズスートの実行時間評価処理



【図３２】

シリーズスートの実行時間評価処理の擬似プログラム

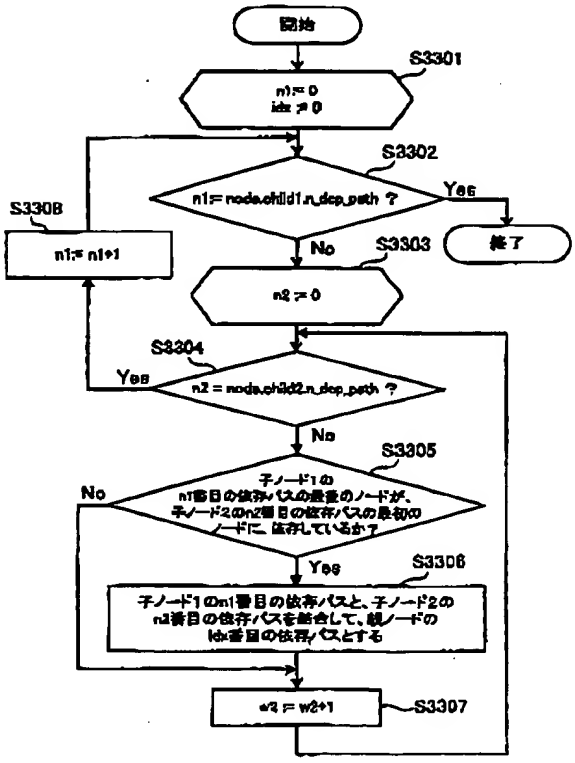
```
procedure series_suite_evaluation(child1, child2, w)
begin
int w, n1, n2, idx=0;

//critical path calculation
for n1=0 to child1.n_dep_path - 1
begin
  for n2=0 to child2.n_dep_path - 1
  begin
    if( child1.dep_path[n1].out  depends on  child2.dep_path[n2].in )
    begin
      concatinate child1.dep_path[n1] & child2.dep_path[n2] as
        node.dep_path[idx];
      node.dep_path[idx].length =
        child1.dep_path[n1].length + child2.dep_path[n2].length
      idx++
    end
  end
end
sort node.dep_path[idx];
node.n = node.dep_path[0];

//best_time calculation
for w=1 to W
begin
  tmp = child1.best_time2[w] + child2.best_time2[w];
  node.best_time[w] = max(node.n, tmp);
  node.best_time2[w] = tmp;
end
end of procedure
```

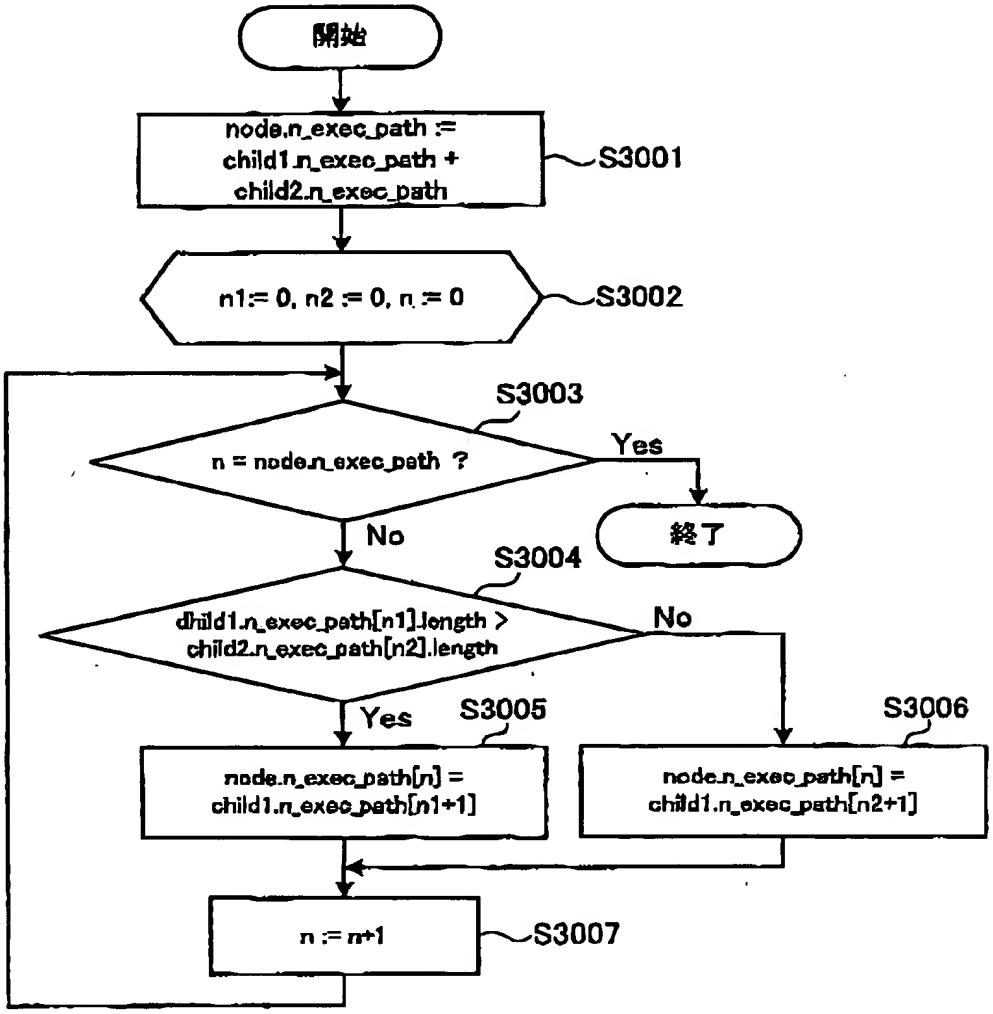
【図３３】

クリティカルパス長再計算処理

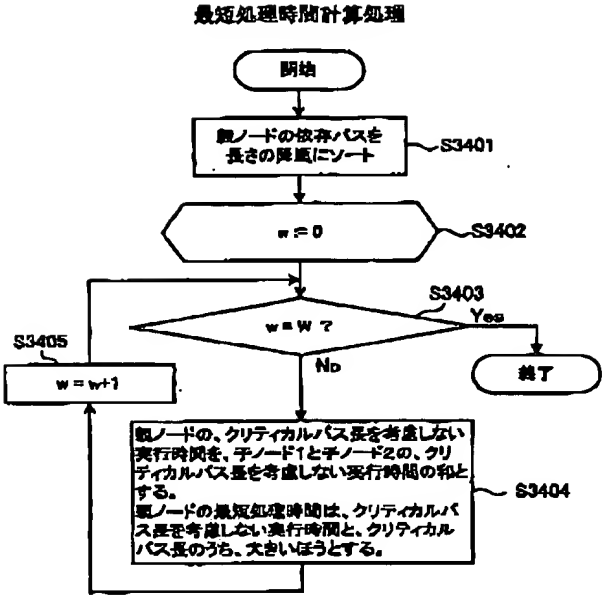


【図30】

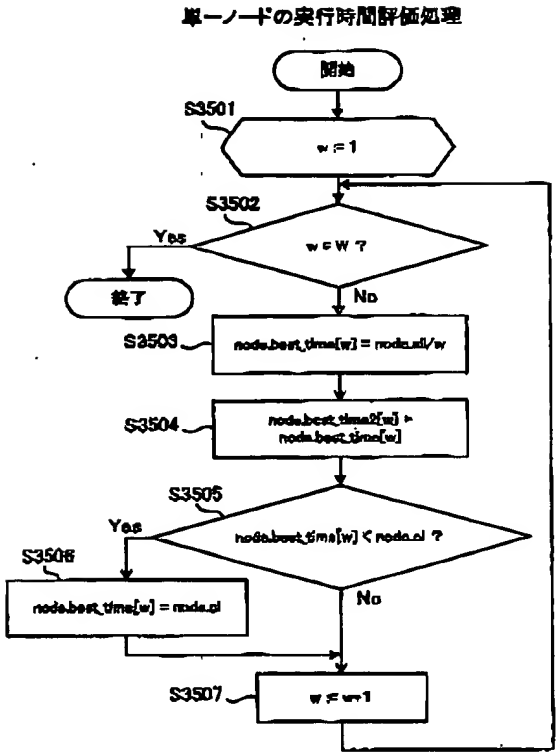
依存パス融合処理



【図３４】



【図３５】



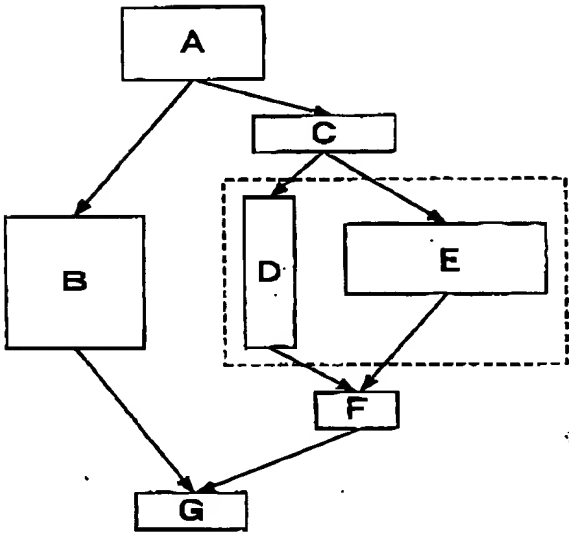
【図３７】

スート/並列度	1	2	3	4	5	6
A	6	3	3	3	3	3
B	12	6	4	4	4	4
C	3	2	1	1	1	1
D	3	3	3	3	3	3
E	8	4	3	2	2	2
F	1	1	1	1	1	1
G	2	1	1	1	1	1
0	36	20	16	13	10	10
1	34	19	15	14	9	9
2	28	16	12	11	6	6
3	16	11	6	6	5	4
4	15	10	5	4	4	4
5	12	8	4	3	3	3

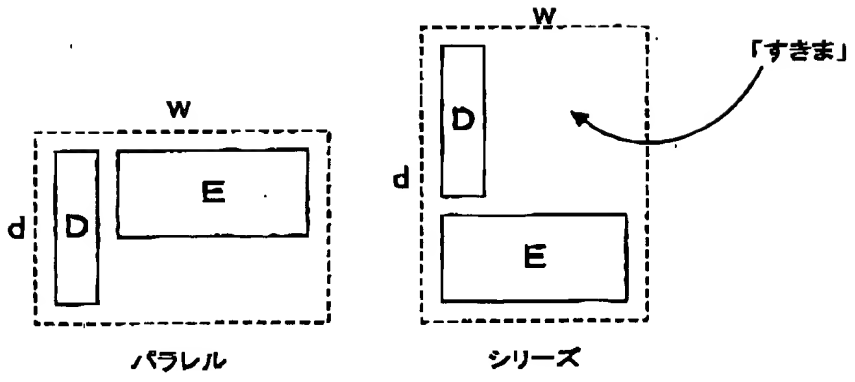
【図３８】

スート/並列度	1	2	3	4	5	6
A	6	3	2	2	1	1
B	12	6	4	3	2	2
C	3	2	1	1	1	1
D	3	2	1	1	1	1
E	8	4	3	2	2	1
F	1	1	1	1	1	1
G	2	1	1	1	1	1
0	36	20	15	14	8	8
1	34	19	14	13	7	7
2	28	16	12	11	6	6
3	16	11	6	6	5	4
4	15	10	5	5	4	3
5	12	8	4	4	3	2

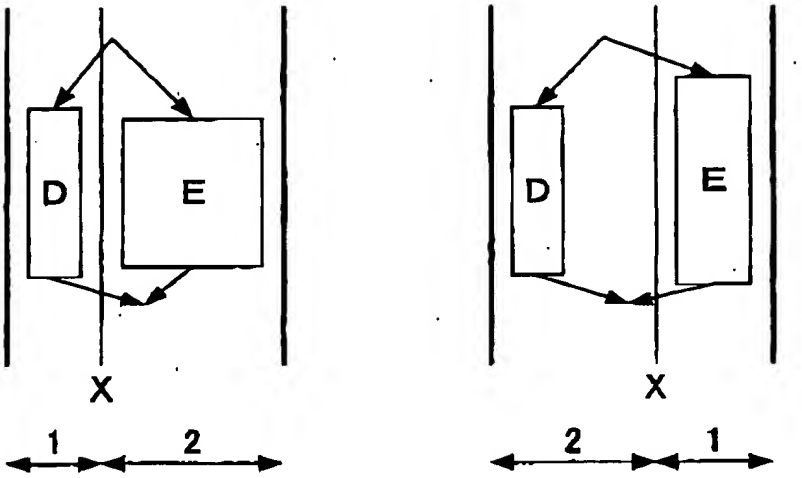
【図39】



【図40】



【図41】



フロントページの続き

(72)発明者 田端 邦男
神奈川県大和市下鶴間1623番地14 日本アイ・ビー・エム株式会社 東京基礎研究所内

(72)発明者 小松 秀昭
神奈川県大和市下鶴間1623番地14 日本アイ・ビー・エム株式会社 東京基礎研究所内

Fターム(参考) 5B013 DD04
5B033 CA22
5B081 CC24 CC32